Introduction à l'analyse syntaxique et à la compilation

Roberto M. Amadio

Introduction à l'analyse syntaxique et à la compilation (notes de cours) *

Roberto M. Amadio
Université Paris Diderot (Paris 7)

25 juin 2009

Résumé

Ces notes de cours sont une introduction aux différentes phases de la compilation des langages de programmation. Après un survol de ces phases basé sur un petit langage d'expressions arithmétiques, on présente la théorie des grammaires algébriques LL et LR. On s'intéresse ensuite à l'évaluation et au typage en s'appuyant sur les techniques de la sémantique opérationnelle. On décline ces techniques dans le cadre de simples langages qui reflètent les styles de programmation impératif, à objets et fonctionnel. Enfin, on aborde les questions de la conception d'une machine virtuelle, de la gestion de la mémoire et de la traduction du langage source dans le langage d'une machine virtuelle.

*Envoyez vos corrections à `amadio@pps.jussieu.fr`

Table des matières

1 Préliminaires

1.1 Objectifs

A l'issu de ce cours, l'étudiant doit être capable :
- de décrire la syntaxe abstraite d'un langage de programmation et de construire un analyseur syntaxique pour ce langage à l'aide d'outils standards.
- de comprendre la description formalisée de la sémantique opérationnelle et de la sémantique statique d'un langage (principalement le typage).
- de programmer un évaluateur et un vérificateur de correction statique.
- d'écrire un compilateur vers une simple machine virtuelle et de mettre en oeuvre la machine virtuelle (notamment la gestion de la mémoire).

1.2 Programme du cours et des travaux pratiques

- Syntaxe des langages de programmation.
- Principes de l'analyse syntaxique descendante (LL) et montante (LR).
- Présentation des outils Lex et Yacc sous OCAML.
- Réalisation d'un analyseur syntaxique complet.
- Évaluation et typage d'un langage impératif, d'un langage à objets et d'un langage fonctionnel.
- Réalisation d'un évaluateur et d'un vérificateur de type pour un langage à objets.
- Machine virtuelle et fonction de compilation pour le langage impératif.
- Principes de gestion de la mémoire.

1.3 Pré-requis

Notions d'algorithmique et de logique. Notions de programmation impérative, à objets et fonctionnelle. Notions de langages formels : langages rationnels et algébriques.

1.4 Bibliographie

Ces notes ne sont qu'une trace synthétique de ce qui est discuté dans le cours. On pourra se référer aux textes suivants pour une présentation plus approfondie.
- A. Appel. *Modern compiler implementation in ML*, Cambridge University Press (en anglais seulement).
- A. Aho, R. Sethi et J. Ullman. *Compilateurs : principes, techniques et outils*. Dunod (disponible aussi en anglais). Un classique, notamment pour la partie sur l'analyse syntaxique.
- R. Wilhelm, D. Maurer. *Les compilateurs— théorie, construction, génération*, Masson (disponible aussi en anglais). Les quatre premiers chapitres de ce texte contiennent des descriptions assez complètes de machines virtuelles pour des langages impératifs, fonctionnels, à objets et logiques.

Ces notes de cours considèrent uniquement des langages de programmation *séquentiels* de type *impératif*, *à objets* et *fonctionnel*. On fait l'impasse sur les questions de compilation liées aux langages parallèles et/ou avec des contraintes de temps réel. Par ailleurs, on ignore les problèmes d'optimisation et de génération de code qui sont associés à la compilation vers

les langages assembleurs. Ces questions comprennent, par exemple, l'optimisation de boucles, l'analyse de vivacité et l'allocation de registres.

2 Introduction

Que fait un ordinateur ? Il exécute des commandes. Que fait un compilateur ? Il traduit des commandes formulées dans un certain langage L (dit langage source) dans les commandes d'un langage L' (dit langage objet) compréhensible par l'ordinateur. Avant de traduire, il faut s'assurer que la 'phrase' à traduire soit bien une phrase du langage source. C'est le but de l'*analyse syntaxique*.

Dans ce cours un langage est toujours un langage de programmation et dans ce contexte on parle de *compilation* plutôt que de *traduction*. Notre premier objectif est d'introduire un petit langage de programmation et de s'en servir pour illustrer de façon informelle les différentes phases de l'analyse syntaxique et de la compilation.

2.1 Le langage source

Le langage source considéré est un langage d'expressions. Il comprend :
- Des constantes qui dénotent des nombres naturels : $0, 12, 345, \ldots$
- Des constantes qui dénotent des valeurs booléennes : true, false.
- Des opérateurs : $+, *, and, or, \ldots$
- Des variables (ou identificateurs) : $x, a45, zzz, \ldots$

Il est aussi possible de déclarer des identificateurs à l'aide d'une instruction *let_in_* et de composer les expressions à l'aide d'une instruction *if_then_else*.

2.2 L'analyse lexicale

La première tâche consiste à identifier les *unités lexicales* de notre langage. Dans notre cas, il s'agira de :
- Mots clefs (*let*, *if*,...)
- Symboles $(+, *, (,), \ldots)$
- Identificateurs (toute suite de lettres et de chiffres qui commence par une lettre).
- Nombres (toute suite de chiffres).

Il faut aussi décider quels sont les symboles qui permettent de séparer les unités lexicales. Typiquement on choisit : le symbole blanc, le retour chariot et la tabulation. Enfin, on utilise souvent un symbole spécial *eof* pour marquer la fin du fichier.

Exemple 2.1 *Dans*

```
let x7=  3
in
 (x7+4 )
eof
```

On devrait identifier les unités lexicales suivantes (qu'on sépare avec un espace) :

```
let x7 = 3 in ( x7 + 4  ) eof
```

La spécification des unités lexicales est effectuée à l'aide d'expressions régulières (on dit aussi rationnelles). Cette approche est particulièrement utile pour les unités lexicales comme

les identificateurs qui comportent une infinité d'éléments. Par exemple, un identificateur est spécifié par les expressions régulières :

$$\begin{aligned} \mathit{chiffre} &= 0 + \cdots + 9 \\ \mathit{lettre} &= a + \cdots + z + A + \cdots + Z \\ \mathit{ident} &= \mathit{lettre} \cdot (\mathit{chiffre} + \mathit{lettre})^* \end{aligned}$$

A partir d'une expression régulière, des algorithmes standards permettent de construire un automate fini qui reconnaît exactement les unités lexicales spécifiées. L'outil LEX permet d'automatiser complètement ce travail. L'utilisateur décrit les unités lexicales et l'outil génère une fonction qui permet de lire la prochaine unité lexicale disponible en entrée ou d'afficher un signal d'erreur si aucune unité lexicale est reconnue.

2.3 L'analyse syntaxique

Grâce à l'analyse lexicale, nous pouvons voir le langage source comme un mot dont les caractères sont les unités lexicales. Ainsi, dans le cas de l'exemple 2.1, on pourrait obtenir un mot de la forme :

$$\mathit{let}\ \mathit{id}(x7)\ \mathit{eq}\ \mathit{cnst}(3)\ \mathit{in}\ \mathit{lpar}\ \mathit{id}(x7)\ \mathit{plus}\ \mathit{cnst}(4)\ \mathit{rpar}\ \mathit{eof}$$

Certains caractères du mot en question comme *id* et *cnst* ont un paramètre. Ce paramètre est une valeur qui est associée à l'unité lexicale. La prochaine tâche consiste à décrire la structure des phrases du langage en s'appuyant sur la théorie des *grammaires non-contextuelles* (ou *algébriques*). Par exemple, voici une grammaire qui décrit notre langage :

$$\begin{aligned} \mathit{op} &\to \mathit{plus} \mid \mathit{prod} \mid \mathit{and} \mid \cdots \\ e &\to \mathit{id} \mid \mathit{cnst} \mid e\ \mathit{op}\ e \mid \mathit{lpar}\ e\ \mathit{rpar} \\ b &\to e \mid \mathit{if}\ e\ \mathit{then}\ b\ \mathit{else}\ b \mid \mathit{let}\ \mathit{id}\ \mathit{eq}\ e\ \mathit{in}\ b \mid \mathit{lpar}\ b\ \mathit{rpar} \\ S &\to b\ \mathit{eof} \end{aligned}$$

Et voici une *dérivation gauche* :

$$\begin{aligned} S &\Rightarrow b\ \mathit{eof} \\ &\Rightarrow \mathit{let}\ \mathit{id}\ \mathit{eq}\ e\ \mathit{in}\ b\ \mathit{eof} \\ &\Rightarrow \mathit{let}\ \mathit{id}\ \mathit{eq}\ \mathit{cnst}\ \mathit{in}\ b\ \mathit{eof} \\ &\Rightarrow \mathit{let}\ \mathit{id}\ \mathit{eq}\ \mathit{cnst}\ \mathit{in}\ e\ \mathit{eof} \\ &\Rightarrow \mathit{let}\ \mathit{id}\ \mathit{eq}\ \mathit{cnst}\ \mathit{in}\ \mathit{lpar}\ e\ \mathit{rpar}\ \mathit{eof} \\ &\Rightarrow \mathit{let}\ \mathit{id}\ \mathit{eq}\ \mathit{cnst}\ \mathit{in}\ \mathit{lpar}\ e\ \mathit{plus}\ e\ \mathit{rpar}\ \mathit{eof} \\ &\Rightarrow \mathit{let}\ \mathit{id}\ \mathit{eq}\ \mathit{cnst}\ \mathit{in}\ \mathit{lpar}\ \mathit{id}\ \mathit{plus}\ e\ \mathit{rpar}\ \mathit{eof} \\ &\Rightarrow \mathit{let}\ \mathit{id}\ \mathit{eq}\ \mathit{cnst}\ \mathit{in}\ \mathit{lpar}\ \mathit{id}\ \mathit{plus}\ \mathit{cnst}\ \mathit{rpar}\ \mathit{eof} \end{aligned}$$

On remarquera que les unités lexicales sont les symboles terminaux de la grammaire. Un premier problème est de déterminer si une certaine suite d'unités lexicales appartient au langage généré par la grammaire. Il y a des algorithmes généraux qui permettent de répondre à cette question mais ces algorithmes ne sont pas très efficaces. Par ailleurs, une grammaire peut être ambiguë et cette ambiguïté est nuisible au processus de compilation. Dans les années 70 on a donc défini un certain nombre de sous-classes des langages algébriques. Les contraintes principales qu'on souhaite satisfaire sont les suivantes :

1. On s'intéresse à des grammaires algébriques *non-ambiguës*.

2. Le langage généré doit être reconnaissable par un automate à pile *déterministe* (APD).
3. La construction de l'APD à partir de la grammaire doit être suffisamment *efficace*.
4. La classe de grammaires considérée doit être suffisamment *expressive* pour traiter les constructions principales des langages de programmation.

L'analyse syntaxique ne se limite pas à déterminer si un mot peut être généré par une grammaire. Deux autres objectifs aussi importants sont :

- Si le mot n'est pas dans le langage généré, il faut produire un message d'erreur informatif qui facilite la correction du programme. Par exemple : *Missing) at line 342.*
- Si le mot peut être généré, alors on veut construire une *représentation interne* du programme qui permettra d'effectuer d'autres vérifications (analyse statique) et éventuellement de procéder à une compilation. On appelle aussi *syntaxe abstraite* cette représentation interne.

2.4 La syntaxe abstraite

La 'syntaxe abstraite' d'un programme est essentiellement un arbre étiqueté. Le fait de passer d'une représentation linéaire (un mot) à une représentation à arbre permet d'éliminer certaines informations syntaxiques comme les parenthèses. Dans des langages à la ML, la représentation d'un arbre étiqueté est directe à l'aide de déclarations de type et de constructeurs. Par exemple, dans notre cas on pourrait avoir :

```
op   = Plus | And | ...
exp  = Id of string | Cnst of int | Op of op * exp * exp
body = Exp of exp | Ite of exp * body * body | Let of string * exp * body
```

Ainsi la syntaxe abstraite associée à notre programme serait :

```
Let("x7", Exp(Cnst(3)), Exp(Op(Plus, Id("x7"), Cnst(4))))
```

Dans d'autres langages comme C ou Java, la représentation d'un arbre étiqueté demande un certain travail de codage (enregistrements, pointeurs,...)

La construction de l'arbre s'effectue en associant des 'actions sémantiques' aux règles de la grammaire. Par exemple, l'action qu'on associe à la règle $b \rightarrow$ *let id eq e in b* pourrait être celle de construire un arbre dont la racine est étiquetée par Let et qui a trois fils qui correspondent aux arbres associés à *id*, *e* et *b*, respectivement.

2.5 L'évaluation

Toute traduction (ou compilation) suppose une préservation de la *signification* (ou sémantique) de la phrase traduite. Comment faire pour décrire la sémantique d'un langage de programmation ? Une idée générale qui s'applique aussi bien aux automates qu'aux langages de programmation est de décrire formellement le calcul et de retenir le 'résultat' du calcul. Cette idée est adéquate au moins pour les langages séquentiels et déterministes que nous traitons dans ce cours.

Dans notre étude de cas, nous pouvons établir que le résultat du calcul d'une expression est une *valeur* v, à savoir ou bien un booléen ou bien un nombre naturel. Pour décrire le calcul on utilise une relation $\Downarrow$. Cette relation est définie sur la syntaxe abstraite mais pour

des raisons de lisibilité on l'écrit toujours en utilisant la syntaxe concrète. La relation est définie par les règles suivantes.

$$\frac{}{v \Downarrow v} \qquad \frac{e_i \Downarrow n_i \in \mathbf{N} \quad i = 1,2}{e_1 \ + \ e_2 \Downarrow n_1 \ + \ n_2}$$

$$\frac{e \Downarrow \mathsf{true} \quad b_1 \Downarrow v}{\mathit{if}\ e\ \mathit{then}\ b_1\ \mathit{else}\ b_2 \Downarrow v} \qquad \frac{e \Downarrow \mathsf{false} \quad b_2 \Downarrow v}{\mathit{if}\ e\ \mathit{then}\ b_1\ \mathit{else}\ b_2 \Downarrow v}$$

$$\frac{e' \Downarrow v' \quad [v'/x]b \Downarrow v}{\mathit{let}\ x = e'\ \mathit{in}\ b \Downarrow v}$$

Ici on se limite au cas où *op* est l'addition sur les nombres naturels.

On dénote par $[v/x]b$ la *substitution* de v pour x dans b qui est définie par récurrence sur la structure de b :

$$\begin{array}{ll}
[v/x]v' & = v' \\
[v/x]y & = \begin{cases} v & \text{si } x = y \\ y & \text{sinon} \end{cases} \\
[v/x](e_1 + e_2) & = [v/x]e_1 + [v/x]e_2 \\
[v/x](\mathit{if}\ e\ \mathit{then}\ b_1\ \mathit{else}\ b_2) & = \mathit{if}\ [v/x]e\ \mathit{then}\ [v/x]b_1\ \mathit{else}\ [v/x]b_2 \\
[v/x](\mathit{let}\ y = e\ \mathit{in}\ b) & = \begin{cases} \mathit{let}\ y = [v/x]e\ \mathit{in}\ b & \text{si } x = y \\ \mathit{let}\ y = [v/x]e\ \mathit{in}\ [v/x]b & \text{sinon} \end{cases}
\end{array}$$

On remarquera que dans $[v/x](\mathit{let}\ x = e\ \mathit{in}\ b)$ l'opération de substitution n'affecte pas b car la variable x est *liée* à l'expression e dans b.

Voici un exemple d'application des règles :

$$\dfrac{3 \Downarrow 3 \qquad \dfrac{4 \Downarrow 4}{[3/x](x+4) \Downarrow 7}}{\mathit{let}\ x = 3\ \mathit{in}\ (x+4) \Downarrow 7}$$

On ne peut pas évaluer une expression qui contient une variable non déclarée. Par exemple, on ne peut pas évaluer $\mathit{let}\ x = 3\ \mathit{in}\ y$. Par ailleurs, le branchement $\mathit{if}\ e\ \mathit{then}\ b_1\ \mathit{else}\ b_2 \Downarrow v$ est défini seulement si le test e s'évalue en une valeur booléenne. Il est toujours possible de *compléter* la définition de la sémantique de façon à prendre en compte ces situations anormales. A cette fin, on introduit la possibilité qu'une expression s'évalue en une nouvelle valeur *err*. On peut

alors reformuler et compléter les règles comme suit :

$$\frac{}{v \Downarrow v} \qquad \frac{e_i \Downarrow n_i \in \mathbf{N} \quad i = 1,2}{e_1 \ + \ e_2 \Downarrow n_1 \ + \ n_2}$$

$$\frac{e_i \Downarrow v_i \quad i = 1,2 \quad \{v_1, v_2\} \not\subset \mathbf{N}}{e_1 \ + \ e_2 \Downarrow \mathit{err}} \qquad \frac{e \Downarrow \mathsf{true} \quad b_1 \Downarrow v}{\mathit{if}\ e\ \mathit{then}\ b_1\ \mathit{else}\ b_2 \Downarrow v}$$

$$\frac{e \Downarrow \mathsf{false} \quad b_2 \Downarrow v}{\mathit{if}\ e\ \mathit{then}\ b_1\ \mathit{else}\ b_2 \Downarrow v} \qquad \frac{e \Downarrow v \quad v \notin \{\mathsf{true}, \mathsf{false}\}}{\mathit{if}\ e\ \mathit{then}\ b_1\ \mathit{else}\ b_2 \Downarrow \mathit{err}}$$

$$\frac{e' \Downarrow v' \quad v' \neq \mathit{err} \quad [v'/x]b \Downarrow v}{\mathit{let}\ x = e'\ \mathit{in}\ b \Downarrow v} \qquad \frac{e' \Downarrow \mathit{err}}{\mathit{let}\ x = e'\ \mathit{in}\ b \Downarrow \mathit{err}}$$

$$\frac{}{x \Downarrow \mathit{err}}$$

2.6 Le typage

Certaines erreurs peuvent être évitées si le programme satisfait des contraintes additionnelles de bonne formation. Une méthode standard pour formuler ces contraintes est de donner des *règles de typage*. Un type est une *abstraction* d'un ensemble de valeurs. Dans notre cas, nous considérons un type *bool* pour l'ensemble des valeurs booléennes $\{\mathsf{true}, \mathsf{false}\}$ et un type *nat* pour l'ensemble des nombres naturels.

Les *jugements* utilisés dans les règles de typage ont la forme $E \vdash b : \tau$ où :

- $\tau \in \{\mathit{bool}, \mathit{nat}\}$ est un type.
- E est un *environnement de type*, c'est-à-dire une fonction partielle à domaine fini des identificateurs aux types.

Il convient d'introduire un minimum de notations :

- On représente par $x_1 : \tau_1, \ldots, x_n : \tau_n$ où $x_i \neq x_j$ si $i \neq j$ l'environnement de type E tel que $E(x_i) = \tau_i$ pour $i = 1, \ldots, n$ et qui n'est pas défini autrement.
- Si $f : D \rightharpoonup D'$ est une fonction partielle, $d \in D$ et $d' \in D'$ alors

$$f[d'/d](x) = \begin{cases} d' & \text{si } x = d \\ f(x) & \text{autrement} \end{cases}$$

  Par exemple, $E[\tau/x]$ est l'environnement de type qui se comporte comme E sauf sur x où il rend le type τ.
- On dénote par $\emptyset$ l'environnement de type à domaine vide et on abrège le jugement $\emptyset \vdash b : \tau$ avec $\vdash b : \tau$.

Les règles de typage sont les suivantes :

$$\frac{n \in \mathbf{N}}{E \vdash n : \mathit{nat}} \qquad \frac{v \in \{\mathsf{true}, \mathsf{false}\}}{E \vdash v : \mathit{bool}} \qquad \frac{E(x) = \tau}{E \vdash x : \tau} \qquad \frac{E \vdash e_i : \mathit{nat} \quad i = 1,2}{E \vdash e_1 + e_2 : \mathit{nat}}$$

$$\frac{E \vdash e : \mathit{bool} \quad E \vdash b_i : \tau \quad i = 1,2}{E \vdash \mathit{if}\ e\ \mathit{then}\ b_1\ \mathit{else}\ b_2 : \tau} \qquad \frac{E \vdash e : \tau' \quad E[\tau'/x] \vdash b : \tau}{E \vdash \mathit{let}\ x = e\ \mathit{in}\ b : \tau}$$

Voici un exemple de preuve de typage.

$$\dfrac{\dfrac{}{\vdash 3 : nat} \qquad \dfrac{\dfrac{x : nat \vdash x : nat \qquad x : nat \vdash 4 : nat}{x : nat \vdash (x + 4) : nat}}{\vdash let\ x = 3\ in\ x + 4 : nat}$$

On remarquera que la valeur *err* n'est pas typable. Notre objectif est de montrer qu'une expression typable ne peut pas s'évaluer à *err*. Comme la valeur *err* n'est pas typable, il suffit de montrer que la relation d'évaluation préserve la typabilité.

Proposition 2.2 *Si* $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash b : \tau$, $\vdash v_i : \tau_i$ *pour* $i = 1, \ldots, n$ *et* $[v_1/x_1, \ldots, v_n/x_n]b \Downarrow v$ *alors* $\vdash v : \tau$.

Idée de la preuve. Par récurrence sur le typage de b. On se limite à traiter le cas le plus intéressant où le programme a la forme $let\ x = e\ in\ b$. On suppose $x \neq x_i$ pour $i = 1, \ldots, n$. Le cas où $x = x_i$ est similaire. Soient $E = x_1 : \tau_1, \ldots, x_n : \tau_n$ et $S = [v_1/x_1, \ldots, v_n/x_n]$. On sait que la preuve de typage termine par :

$$\dfrac{E \vdash e : \tau' \quad E[\tau'/x] \vdash b : \tau}{E \vdash let\ x = e\ in\ b : \tau}$$

Par ailleurs, on sait que :

$$S(let\ x = e\ in\ b) = (let\ x = Se\ in\ Sb) \Downarrow v$$

Par définition de l'évaluation, on doit avoir $Se \Downarrow v'$. Par hypothèse de récurrence sur $E \vdash e : \tau'$, il suit que $\vdash v' : \tau'$. Donc $v' \neq err$. Mais alors on doit avoir : $S[v'/x]b \Downarrow v$. Par hypothèse de récurrence sur $E[\tau'/x] \vdash b : \tau$, il suit que $\vdash v : \tau$. •

Exercice 2.3 *Montrez qu'il y a des expressions qui ne s'évaluent pas en erreur et qui ne sont pas typables.*

Exercice 2.4 *Démontrez une version plus forte de la proposition 2.2, à savoir si* $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash b : \tau$ *et* $\vdash v_i : \tau_i$ *pour* $i = 1, \ldots, n$ *alors* $[v_1/x_1, \ldots, v_n/x_n]b \Downarrow v$ *et* $\vdash v : \tau$. *Cette version 'forte' ne se généralise pas à des langages plus compliqués car souvent le typage ne suffit pas à assurer la terminaison du programme évalué.*

2.7 Une machine virtuelle et son langage

Une *machine virtuelle* est un ensemble de structures de données et d'algorithmes qui permettent d'exécuter (efficacement) un certain ensemble d'instructions. En d'autres termes, une machine virtuelle peut être vue comme un *type de donnée*.

On considère une machine virtuelle dont la *mémoire* est divisée en deux parties :

- Une zone *statique* qui contient un compteur ordinal *pc*, un pointeur à la pile *sp* et les instructions du programme à exécuter.
- Une *pile de valeurs* dont le sommet est pointé par *sp*.

On voit la mémoire comme un tableau M et on dénote, par exemple, par $M[sp]$ la cellule du tableau d'adresse sp. La machine peut exécuter un certain nombre d'instructions dont nous décrivons l'effet sur les structures de la machine virtuelle :

build v	$sp := sp + 1; M[sp] := v; pc := pc + 1$
branch j	(si $M[sp] =$ true alors $pc := pc + 1$ sinon $pc := j$); $sp := sp - 1$
load i	$sp := sp + 1; M[sp] := M[i]; pc := pc + 1$
add	$sp := sp - 1; M[sp] := M[sp] + M[sp + 1]; pc := pc + 1$
return	$pc := 0; M[0] := M[sp]; sp := 0$

Le cycle de chargement exécution (*fetch and execute*) est :

$$\begin{array}{l} pc := 1; sp := 0; \\ \mathit{while}\ pc \neq 0\ \mathit{do} \\ \mathit{execute}(pc) \end{array}$$

Donc le calcul termine quand $pc = 0$ et dans ce cas le résultat est à l'adresse 0 de la pile.

On remarquera que la machine virtuelle *ne vérifie pas les types des données*. Ainsi il est possible d'exécuter une instruction branch lorsque $M[sp]$ ne contient pas une valeur booléenne ou d'exécuter une instruction add lorsque le sommet de la pile ne contient pas deux nombres naturels. On pourrait ajouter des actions qui vérifient les types au moment de l'exécution. Alternativement, on pourrait s'assurer statiquement que le code compilé ne produit jamais des erreurs de type.

2.8 La compilation

On souhaite compiler un programme b dans une liste d'instructions de la machine virtuelle. On définit une fonction de compilation $\mathcal{C}$ qui prend en paramètre une expression et une liste de variables w et produit une liste d'instructions de la machine virtuelle. La liste w garde une trace des déclarations *let* $x = \ldots$ et naturellement au début de la compilation on suppose que la liste est vide. Nous utilisons la notation $i(x, w)$ pour indiquer la position la plus à droite de l'occurrence de x dans w. Par exemple $i(x, y \cdot x \cdot z \cdot x \cdot y) = 4$.

$$\begin{array}{ll}
\mathcal{C}(x, w) & = (\mathsf{load}\ i(x, w)) \cdot (\mathsf{return}) \\
\mathcal{C}(v, w) & = (\mathsf{build}\ v) \cdot (\mathsf{return}) \\
\mathcal{C}(e_1 + e_2, w) & = \mathcal{C}'(e_1, w) \cdot \mathcal{C}'(e_2, w) \cdot (\mathsf{add}) \cdot (\mathsf{return}) \\
\mathcal{C}'(x, w) & = \mathsf{load}\ i(x, w) \\
\mathcal{C}'(v, w) & = \mathsf{build}\ v \\
\mathcal{C}'(e_1 + e_2, w) & = \mathcal{C}'(e_1, w) \cdot \mathcal{C}'(e_2, w) \cdot (\mathsf{add}) \\
\mathcal{C}(\mathit{let}\ x = e\ \mathit{in}\ b, w) & = \mathcal{C}'(e, w) \cdot \mathcal{C}(b, w \cdot x) \\
\mathcal{C}(\mathit{if}\ e\ \mathit{then}\ b_1\ \mathit{else}\ b_2, w) & = \mathcal{C}'(e, w) \cdot (\mathsf{branch}\ \kappa) \cdot \mathcal{C}(b_1, w) \cdot \kappa : \mathcal{C}(b_2, w)
\end{array}$$

On introduit une fonction auxiliaire $\mathcal{C}'$ qui ne retourne pas le résultat à la fin du calcul. Dans la dernière équation, nous utilisons la notation $\kappa : \mathcal{C}(\ldots)$ pour indiquer que l'adresse de la première instruction de $\mathcal{C}(\ldots)$ est κ.

Exemple 2.5 *La compilation de l'expression*

$$\begin{array}{l}
\mathit{let}\ x = 3\ \mathit{in} \\
\quad \mathit{let}\ y = x + x\ \mathit{in} \\
\quad\quad \mathit{let}\ x = \mathsf{true}\ \mathit{in} \\
\quad\quad\quad \mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ x
\end{array}$$

est la suivante où dans la troisième colonne on présente le contenu de la pile au moment de l'exécution :

1 :	build 3	_
2 :	load 1	3
3 :	load 1	3 3
4 :	add	3 3 3
5 :	build true	3 6
6 :	load 3	3 6 true
7 :	branch 10	3 6 true true
8 :	load 2	3 6 true
9 :	return	3 6 true 6
10 :	load 3	*(code mort)*
11 :	return	*(code mort)*

On remarquera que certaines instructions sont redondantes ou inaccessibles (code mort).

Exercice 2.6 *Considérez :*

$$
\begin{array}{l}
b = \mathit{let}\ x = 3\ \mathit{in} \\
\quad \mathit{let}\ x = x + x\ \mathit{in} \\
\quad\quad \mathit{let}\ y = \mathsf{true}\ \mathit{in} \\
\quad\quad\quad \mathit{if}\ y\ \mathit{then}\ x + x\ \mathit{else}\ x
\end{array}
$$

Calculez les unités lexicales, l'arbre de dérivation et la syntaxe abstraite de b*. Évaluez et typez* b*. Compilez* b *et exécutez le code compilé.*

Un point important que nous omettons de traiter ici est celui de la *correction* de la fonction de compilation. Un premier pas pourrait consister à montrer que la compilation d'un programme bien typé ne produit pas d'erreurs au moment de l'exécution. Ensuite, on pourrait chercher à montrer que si un programme bien typé s'évalue dans une valeur v alors l'exécution du code compilé produira comme résultat la valeur v.

3 Grammaires Algébriques (rappel)

3.1 Grammaires

Définition 3.1 *Une grammaire G est un vecteur (V, Σ, S, R) où V est un ensemble fini de symboles terminaux et non-terminaux, $\Sigma \subseteq V$ est l'ensemble des symboles terminaux, $S \in V \backslash \Sigma$ est le symbole initial (et un non-terminal), et $R \subseteq_{fin} V^+ \times V^*$ est l'ensemble des règles (ou productions).*

Par convention, on utilise $a, b, \ldots$ pour les symboles terminaux, $u, v, \ldots$ pour les mots sur Σ, $A, B, \ldots, S$ pour les symboles non-terminaux, $X, Y, \ldots$ pour les symboles terminaux et non-terminaux et $\alpha, \beta, \gamma, \ldots$ pour les mots sur V. On écrit $(\alpha, \beta) \in R$ comme $\alpha \to \beta$. La notation $\alpha \to \beta_1 \mid \cdots \mid \beta_n$ est une abréviation pour $\alpha \to \beta_1, \ldots, \alpha \to \beta_n$.

Une *configuration* est simplement un mot sur V. Si $\alpha, \beta \in V^*$ sont des configurations alors $\alpha \Rightarrow \beta$ (α se ré-écrit dans β dans une étape) si

$$\alpha = \gamma_1\gamma\gamma_2, \quad \gamma \to \gamma' \quad \text{et} \qquad \beta = \gamma_1\gamma'\gamma_2 \ .$$

On dénote par $\Rightarrow^*$ la clôture réflexive et transitive de $\Rightarrow$. Le langage $\mathcal{L}(G)$ *généré* par une grammaire G est défini par

$$\mathcal{L}(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\} \ .$$

Dans les années 60, Noam Chomsky a proposé une classification des grammaires selon le type des règles utilisées. En particulier :

Linéaires droites Les règles ont la forme $A \to w$ ou $A \to wB$ avec $A, B \in V \backslash \Sigma$ et $w \in \Sigma^*$. Ces grammaires génèrent exactement les langages réguliers (ou rationnels).

Non contextuelles Les règles ont la forme $A \to \alpha$ avec $A \in V \backslash \Sigma$ et $\alpha \in V^*$. Ces grammaires génèrent exactement les langages non-contextuels (on dit aussi algébriques).

Exemple 3.2 *La grammaire suivante décrit les mots sur l'alphabet $\{a, b\}$ qui peuvent être lus indifféremment de gauche à droite et de droite à gauche.*

$$S \to \epsilon \mid a \mid b \mid aSa \mid bSb \ .$$

La grammaire est non-contextuelle mais elle n'est pas linéaire droite.

3.2 Dérivations et ambiguïté

Soit $G = (V, \Sigma, S, R)$ une grammaire non-contextuelle. Une *dérivation* est une séquence de pas de réécriture qui mène du symbole S à un mot $w \in \Sigma^*$. Une dérivation gauche (droite) est une dérivation où dans tous les pas de réécriture on développe le symbole non-terminal le plus à gauche (droite).

Une dérivation peut être représentée par un *arbre de dérivation* dont les noeuds sont étiquetés par les symboles dans V. La racine de l'arbre est étiquetée avec le symbole initial S. Quand une règle $A \to X_1 \cdots X_n$ avec $X_i \in V$ est appliquée, on ajoute n fils au noeud qui correspond à A et on les étiquette avec $X_1, \ldots, X_n$. On remarque qu'il peut y avoir plusieurs dérivations qui produisent le même arbre mais que les dérivations gauches (droites) sont en correspondance bijective avec les arbres de dérivation.

Définition 3.3 (ambiguïté) *Une grammaire est ambiguë s'il y a un mot $w \in \mathcal{L}(G)$ qui admet deux arbres de dérivation différents. Un langage non-contextuel est ambigu si toutes les grammaires non-contextuelles qui le génèrent sont ambiguës.*

Exemple 3.4 *Il est facile de vérifier que la grammaire $E \to i \mid E + E \mid E * E$ est ambiguë. Il est beaucoup plus difficile de montrer que le langage $\{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$ est ambigu.*

Exercice 3.5 *Soit $G = (V, T, S, R)$ une grammaire non-contextuelle où $V = \{S, A, B, a, b\}$, $T = \{a, b\}$ et*

$$R = \{S \to \epsilon \mid aB \mid bA, \quad A \to aS \mid bAA \mid a, \quad B \to bS \mid aBB \mid b\}.$$

1. *Donnez un arbre de dérivation, ainsi que les dérivations gauches et droites de la chaîne aaabbabbba.*
2. *G est-elle ambiguë ?*

3.3 Simplification de grammaires non-contextuelles

Définition 3.6 *Soit $G = (V, \Sigma, S, R)$ une grammaire non-contextuelle. Un symbole $X \in V$ est* utile *si $\exists\, w \in \Sigma^*$ $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$.*

Proposition 3.7 *Soit $G = (V, \Sigma, S, R)$ une grammaire non-contextuelle telle que $\mathcal{L}(G) \neq \emptyset$. Alors on peut construire une grammaire non-contextuelle équivalente G' telle que tous les symboles sont utiles.*

Idée de la preuve. On procède en trois étapes.

(1) D'abord on montre qu'on peut éliminer tous les symboles non-terminaux qui ne peuvent pas produire un mot de symboles terminaux. A cette fin, on calcule itérativement le plus petit ensemble U tel que : (i) si $A \to w$ alors $A \in U$ et (ii) si $A \to X_1 \dots X_n$ et $X_i \in \Sigma \cup U$ pour tout $i = 1, \dots, n$ alors $A \in U$. Ensuite, on élimine toutes les règles qui contiennent des symboles non-terminaux qui ne sont pas dans U.

(2) On calcule itérativement l'ensemble des symboles non-terminaux qui sont accessible à partir du symbole initial. Soit U le plus petit ensemble tel que (i) $S \in U$, et (ii) si $A \to \alpha$ et $A \in U$ alors les non-terminaux dans α sont dans U. On élimine toutes les règles qui contiennent des symboles non-terminaux qui ne sont pas dans U.

(3) En supposant (1) et (2), on peut aussi déterminer les symboles terminaux utiles. Ces sont les symboles qui paraissent à droite d'une des règles restantes. •

Exemple 3.8 *On considère la grammaire :*

$$S \to AB \mid a \quad A \to a$$

En appliquant la méthode précédente on arrive à la grammaire $S \to a$. Remarquez que le résultat n'est pas celui souhaité si on applique la simplification (2) avant la simplification (1).

Définition 3.9 *Une ϵ-règle est une règle de la forme $A \to \epsilon$.*

Proposition 3.10 *Soit $G = (V, \Sigma, S, R)$ une grammaire non-contextuelle. Alors :*

(1) *On peut déterminer si $\epsilon \in \mathcal{L}(G)$.*

(2) *Si $\epsilon \notin \mathcal{L}(G)$ alors on peut construire une grammaire non-contextuelle équivalente sans ϵ-règles.*

(3) *Si $\epsilon \in \mathcal{L}(G)$ alors on peut construire une grammaire non-contextuelle équivalente qui contient exactement une ϵ-règle de la forme $S \to \epsilon$ où S est le symbole initial et tel que le symbole initial S ne paraît pas à droite d'une règle.*

IDÉE DE LA PREUVE. (1) On calcule itérativement les symboles non-terminaux qui peuvent se réduire au mot ϵ.

(2) Pour chaque règle $A \to X_1 \cdots X_n$ on génère les sous-règles $A \to X_{i_1} \cdots X_{i_k}$ où $1 \leq i_1 < \cdots < i_k \leq n$, $k \geq 1$ et les éléments effacés X_j se ré-écrivent en ϵ.

(3) Si $\epsilon \in \mathcal{L}(G)$ alors la méthode décrite dans (2) produit une grammaire $G = (V, \Sigma, S, R)$ qui génère le langage $\mathcal{L}(G)\backslash\{\epsilon\}$. Ensuite on ajoute un nouveau symbole initial S' et les règles $S' \to \epsilon \mid S$. •

Exemple 3.11 *Considérons la grammaire $S \to aS'b$, $S' \to \epsilon \mid aS'b$. L'élimination de la ϵ-règle mène à la grammaire : $S \to ab \mid aS'b$, $S' \to ab \mid aS'b$.*

Exercice 3.12 *Soit G une grammaire non-contextuelle sans ϵ-règles. Montrez qu'on peut éliminer les règles de la forme $A \to B$ où A, B sont des symboles non-terminaux.*

3.4 Automates à pile

Un automate à pile (AP) est un automate fini qui dispose en plus d'une pile. Les automates à pile reconnaissent exactement les langages non-contextuels.

Définition 3.13 *Un automate à pile M est un vecteur $(\Sigma, Q, q_o, F, \Gamma, Z_o, \delta)$ où Σ est l'alphabet d'entrée, Q est un ensemble fini d'états, $q_o \in Q$ est l'état initial, $F \subseteq Q$ est l'ensemble des états finaux, Γ est l'alphabet de la pile, $Z_o \in \Gamma$ est le symbole initialement présent sur la pile et enfin*

$$\delta : (\Sigma \cup \{\epsilon\}) \times Q \times \Gamma \to 2^{(Q \times \Gamma^*)}$$

est la fonction de transition.

Une *configuration* est un triplet $(w, q, \gamma) \in \Sigma^* \times Q \times \Gamma^*$. La relation de *réduction* est définie par

$$\begin{array}{ll} (aw, q, Z\gamma) \vdash_M (w, q', \gamma'\gamma) & \text{si } (q', \gamma') \in \delta(a, q, Z) \\ (w, q, Z\gamma) \vdash_M (w, q', \gamma'\gamma) & \text{si } (q', \gamma') \in \delta(\epsilon, q, Z) \end{array}$$

Donc M placé dans l'état q et avec le symbole Z au sommet de la pile peut se déplacer dans l'état q' en remplaçant Z par γ' au sommet de la pile et en lisant (ou sans lire) un symbole de l'entrée. On remarque qu'à chaque pas un AP *doit* lire un symbole de la pile et que donc le calcul s'arrête si la pile est vide.

Le langage reconnu par un AP M est défini par

$$\mathcal{L}(M) = \{w \in \Sigma^* \mid \exists\, \gamma \in \Gamma^*, q \in F \;\; (w, q_o, Z_o) \vdash_M^* (\epsilon, q, \gamma)\} \; .$$

Dans ce cas on dit que l'AP accepte sur *état final*. Une définition alternative consiste à accepter sur *pile vide*. Formellement

$$\mathcal{L}(M) = \{w \in \Sigma^* \mid \exists\, q \in Q \;\; (w, q_o, Z_o) \vdash_M^* (\epsilon, q, \epsilon)\} \; .$$

Si on prend la deuxième définition, il est inutile de spécifier l'ensemble F d'états finaux dans la définition d'AP. Il n'est pas trop difficile de montrer qu'un langage est accepté par un AP sur état final si et seulement si il est accepté par un AP (pas forcément le même) sur pile vide.

Un AP peut aussi être représenté comme un graphe dirigé : une arête de l'état q à l'état q' est étiquetée par le triplet $(a, X)/\gamma$ ssi $(q', \gamma) \in \delta(a, q, X)$, où $a \in \Sigma \cup \{\epsilon\}$.

Proposition 3.14 *Les langages reconnus par les AP sont exactement ceux générés par les grammaires non-contextuelles.*

3.5 Automates à pile déterministes

Définition 3.15 (APD) *Un AP déterministe (APD) est un AP tel que chaque configuration accessible a au plus un successeur immédiat.*

Proposition 3.16 *Les conditions suivantes assurent qu'un AP* $M = (\Sigma, Q, q_o, F, \Gamma,\ Z_o, \delta)$ *est déterministe :*

(1) $\sharp\delta(a, q, X) \leq 1$ *pour* $a \in \Sigma \cup \{\epsilon\}$.

(2) *Si* $\delta(\epsilon, q, X) \neq \emptyset$ *alors* $\forall\, a \in \Sigma \;\; \delta(a, q, X) = \emptyset$.

Les langages reconnus par les APD forment une classe intermédiaire entre les langages réguliers et les langages non-contextuels. En particulier, il n'est pas possible de déterminiser les AP. Les langages reconnus par les APD sont stables par complémentaire mais pas par union (ou intersection).

Exemple 3.17 *Il n'y a pas d'APD qui puisse reconnaître le langage* $\{a^i b^j c^k \mid i = j \;\; ou \;\; j = k\}$.

Exercice 3.18 *On considère la grammaire* G

$$S \rightarrow aSbS \mid bSaS \mid \epsilon \; .$$

Soit L *le langage généré par* G *et* $L' = L \backslash \{\epsilon\}$. *Montrez que :*

1. *La grammaire* G *est ambiguë.*
2. *Il y a une grammaire sans* ϵ*-règles qui génère le langage* L'.

Exercice 3.19 *Construisez un AP déterministe qui reconnaît le langage généré par la grammaire :*

$S' \rightarrow S\$$

$S \rightarrow if\ E\ then\ S\ else\ S$	$L \rightarrow end$
$S \rightarrow begin\ S\ L$	$L \rightarrow ; SL$
$S \rightarrow print\ E$	$E \rightarrow num\ =\ num$

4 Grammaires LL

On considère une première classe de langages non-contextuels, dits langages LL (pour *left-to-right parse, leftmost derivation*), qui sont reconnus par un APD. L'APD cherche à construire une dérivation gauche en gardant sur la pile les symboles qui doivent encore être ré-écrits. Au début du calcul le symbole initial S est placé sur la pile. A chaque pas, l'automate décide (de façon déterministe) quelle règle appliquer simplement en regardant les k premiers symboles en entrée et le symbole au sommet de la pile. Nous considérons ici le cas $k = 1$.

4.1 Fonctions *First* et *Follow*

Définition 4.1 *Soit $G = (V, \Sigma, S, R)$ une grammaire non-contextuelle, $A \in V \backslash \Sigma$ et $\gamma \in V^*$. On définit :*

(1) *$null(A)$ si $A \Rightarrow^* \epsilon$.*

(2) *$First(\gamma) = \{a \in \Sigma \mid \exists\, \alpha \;\; \gamma \Rightarrow^* a\alpha\}$.*

(3) *$Follow(A) = \{a \in \Sigma \mid S \Rightarrow^* \alpha A a \beta\}$.*

Donc si $a \in First(\gamma)$ alors on peut récrire γ dans un mot dont le premier symbole est a. D'autre part si $a \in Follow(A)$ alors on peut trouver une dérivation où a suit immédiatement A.

On suppose que tous les symboles sont *utiles* (cf. section 3). Nous avons déjà considéré le calcul du prédicat *null* (cf. section 3). Clairement $First(\epsilon) = \emptyset$ et si $a \in \Sigma$ alors $First(a) = \{a\}$. De plus,

$$First(X_1 \dots X_n) = \bigcup_{i=1,\dots,n} \{First(X_i) \mid null(X_1), \dots, null(X_{i-1})\} \ .$$

Il est donc suffisant de déterminer *First* sur les symboles non-terminaux. La fonction *First* est la plus petite fonction (au sens de l'inclusion ensembliste) telle que pour toutes les règles $A \to Y_1 \dots Y_n X \alpha$, $n \geq 0$, $X \in V$, on a

$$null(Y_i), 1 \leq i \leq n \quad \Rightarrow \quad First(A) \supseteq First(X) \ .$$

La fonction *Follow* est la plus petite fonction telle que si $A \to \alpha B Y_1 \dots Y_n X \beta$, $n \geq 0$, $X \in V$ alors

$$null(Y_i), 1 \leq i \leq n \quad \Rightarrow \quad Follow(B) \supseteq First(X)$$

et si $A \to \alpha B Y_1 \dots Y_n$, $n \geq 0$ alors

$$null(Y_i), 1 \leq i \leq n \quad \Rightarrow \quad Follow(B) \supseteq Follow(A) \ .$$

4.2 Grammaires LL(1)

Définition 4.2 *Une grammaire non-contextuelle $G = (V, \Sigma, S, R)$ est LL(1) si pour tous les symboles non-terminaux A avec règles $A \to \alpha_1 \mid \dots \mid \alpha_n$ les conditions suivantes sont satisfaites :*

(1) *$First(\alpha_i) \cap First(\alpha_j) = \emptyset$ si $i \neq j$.*

(2) *Si $null(A)$ alors $First(\alpha_i) \cap Follow(A) = \emptyset$ pour $i = 1, \dots, n$.*

Exemple 4.3 *Considérez la grammaire*

$$S \to d \mid ABS \quad A \to B \mid a \quad B \to \epsilon \mid c$$

Alors

	null	*First*	*Follow*
S	*non*	$\{a, c, d\}$	$\emptyset$
A	*oui*	$\{a, c\}$	$\{a, c, d\}$
B	*oui*	$\{c\}$	$\{a, c, d\}$

La grammaire n'est pas LL(1) car par exemple $null(A)$, $A \to a$ *et* $a \in Follow(A) \cap First(a)$.

Sans perte de généralité, on peut supposer que si $null(A)$ alors $A \to \epsilon$ est une règle de la grammaire. Si la grammaire est LL(1) alors on associe au couple $(a, A) \in \Sigma \times (V \backslash \Sigma)$ une règle comme suit :

– Si $a \in First(\alpha_i)$ alors on applique la règle $A \to \alpha_i$.

– Si $null(A)$ et $a \in Follow(A)$ alors on choisit la règle $A \to \epsilon$.

Ensuite on peut construire un APD $M = (\Sigma, \{q_o\} \cup \{q_a \mid a \in \Sigma\}, q_o, V, S, \delta)$ dont la fonction de transition δ est définie par :

$$\begin{array}{ll}
\delta(a, q_o, a) = \{(q_o, \epsilon)\} & \text{si } a \in \Sigma \\
\delta(a, q_o, A) = \{(q_a, \alpha)\} & \text{si } a \in \Sigma, A \to \alpha, a \in First(\alpha) \\
\delta(a, q_o, A) = \{(q_a, \epsilon)\} & \text{si } a \in \Sigma, null(A), a \in Follow(A) \\
\delta(\epsilon, q_a, a) = \{(q_o, \epsilon)\} & \text{si } a \in \Sigma \\
\delta(\epsilon, q_a, A) = \{(q_a, \alpha)\} & \text{si } a \in \Sigma, A \to \alpha, a \in First(\alpha) \\
\delta(\epsilon, q_a, A) = \{(q_a, \epsilon)\} & \text{si } a \in \Sigma, null(A), a \in Follow(A)
\end{array}$$

Exemple 4.4 *Considérez la grammaire*

$$S \to iEtSeS \mid c \quad E \to b$$

On dérive

	null	*First*	*Follow*
S	*non*	$\{i, c\}$	$\{e\}$
E	*non*	$\{b\}$	$\{t\}$

Donc la grammaire est LL(1) et on peut construire un APD qui reconnaît le langage généré. L'APD a comme alphabet d'entrée $\Sigma = \{i, t, e, c, b\}$ *et les transitions suivantes :*

$$\begin{array}{ll}
(q_o, x, x) \mapsto (q_o, \epsilon) \quad x \in \Sigma & (q_o, c, S) \mapsto (q_c, c) \\
(q_o, i, S) \mapsto (q_i, iEtSeS) & (q_o, b, E) \mapsto (q_b, b) \\
(q_c, \epsilon, c) \mapsto (q_o, \epsilon) & (q_c, \epsilon, S) \mapsto (q_c, c) \\
(q_i, \epsilon, i) \mapsto (q_o, \epsilon) & (q_i, \epsilon, S) \mapsto (q_i, iEtSeS) \\
(q_b, \epsilon, E) \mapsto (q_b, b) & (q_b, \epsilon, b) \mapsto (q_o, \epsilon)
\end{array}$$

Voici le calcul par lequel l'APD reconnaît l'entrée ibtcec avec pile vide :

$$\begin{array}{llll}
(q_o, ibtcec, S) & \vdash (q_i, btcec, iEtSeS) & \vdash (q_o, btcec, EtSeS) & \vdash (q_b, tcec, btSeS) \\
\vdash (q_o, tcec, tSeS) & \vdash (q_o, cec, SeS) & \vdash (q_c, ec, ceS) & \vdash (q_o, ec, eS) \\
\vdash (q_o, c, S) & \vdash (q_c, \epsilon, c) & \vdash (q_o, \epsilon, \epsilon) &
\end{array}$$

Exemple 4.5 *Considérez la grammaire*

$$S \to S'\$ \quad S' \to A \mid B \mid \epsilon \quad A \to aAb \mid \epsilon \quad B \to bBa \mid \epsilon$$

On dérive :

	null	*First*	*Follow*
S	*non*	$\{\$, a, b\}$	$\emptyset$
S'	*oui*	$\{a, b\}$	$\{\$\}$
A	*oui*	$\{a\}$	$\{b, \$\}$
B	*oui*	$\{b\}$	$\{a, \$\}$

Donc la grammaire est LL(1) et on peut construire un APD qui reconnaît le langage généré.

Exercice 4.6 *Expliquez pourquoi la grammaire suivante n'est pas LL(1) :*

$$S \to iEtSS' \mid a \qquad S' \to eS \mid \epsilon \qquad E \to b$$

La grammaire est-elle ambiguë ?

Exercice 4.7 *On considère la grammaire suivante avec symbole initial S :*

$$\begin{array}{ll} S \to G\$ & G \to P \\ G \to PG & P \to id : R \\ R \to \epsilon & R \to idR \end{array}$$

1. *Calculez les fonctions null, First et Follow sur les symboles non-terminaux de la grammaire.*
2. *La grammaire est-elle LL(1) ? Motivez votre réponse.*

Exercice 4.8 *On considère la grammaire.*

$$S \to A \mid B \quad A \to aAb \mid 0 \quad B \to aBbb \mid 1$$

Calculer le First de S, A et B. La grammaire est-elle LL(1) ? Pouvez vous construire un automate à pile déterministe qui reconnaît le langage généré par la grammaire ?

Exercice 4.9 *On considère la grammaire :*

$$\begin{array}{llll} S & \to E\$ & E & \to TE' \\ E' & \to +TE' \mid \epsilon & T & \to FT' \\ T' & \to *FT' \mid \epsilon & F & \to (E) \mid id \end{array}$$

1. *Calculez First des symboles non terminaux.*
2. *Calculez Follow des symboles non terminaux.*
3. *La grammaire est-elle LL(1) ? Expliquez.*
4. *Calculez une grammaire équivalente sans ϵ-règles.*

Exercice 4.10 *Le but de cet exercice est de montrer que l'on peut toujours éliminer une production de la forme $A \to A\alpha$ (production qu'on appelle récursive gauche). Cette élimination peut aider à construire une grammaire LL équivalente. Dans la suite on appelle A-production une production de la forme $A \to \beta$. Soit G une grammaire,*

$$A \to A\alpha_1 \mid \ldots \mid A\alpha_n$$

l'ensemble des A-productions ayant A comme symbole le plus à gauche et

$$A \to \beta_1 \mid \ldots \mid \beta_m$$

les autres A-productions. Considérez la grammaire G' obtenue en ajoutant un symbole B et en remplaçant les A-productions par :

$$A \to \beta_1 \mid \ldots \mid \beta_m \mid \beta_1 B \mid \ldots \mid \beta_m B$$

et

$$B \to \alpha_1 \mid \ldots \mid \alpha_n \mid \alpha_1 B \mid \ldots \mid \alpha_n B$$

Montrez que G' est équivalente à G.

Exercice 4.11 *On considère la grammaire suivante :*

$$\begin{array}{ll} S \to E\$ & E \to E + T \mid E - T \mid T \\ T \to T * F \mid T/F \mid F & F \to id \mid num \mid (E) \end{array}$$

Éliminez les productions récursives gauches.

5 Grammaires LR

Dans les APD associés aux grammaires LL on cherche à construire une dérivation gauche à partir du symbole initial. On procède de la racine vers les feuilles et on effectue des *expansions* de symboles non-terminaux, c'est-à-dire on remplace la partie gauche d'une règle (un non-terminal) par sa partie droite.

Dans les APD associés aux grammaires LR (pour *left-to-right parse, rightmost derivation*) on cherche à construire une dérivation droite à partir des feuilles. On procède des feuilles vers la racine et on effectue des *réductions*, c'est-à-dire on remplace la partie droite d'une règle par sa partie gauche (un non-terminal). Plus précisément, l'APD peut effectuer deux types d'actions :

shift On déplace un symbole de l'entrée vers la pile.

reduce On remplace un mot α au sommet de la pile par un non-terminal X, à condition que $X \to \alpha$ soit une règle de la grammaire.

L'APD peut lire k symboles de l'entrée avant de décider quelle action exécuter.

Exemple 5.1 *On considère la grammaire :*

$$\begin{array}{ll} S' \to S\$ & S \to S; S \mid id := E \mid print(L) \\ E \to id \mid num \mid E + E \mid (S, E) & L \to E \mid L, E \end{array}$$

Il se trouve que cette grammaire est LR. Dans la table 1, nous décrivons l'exécution de l'APD où on omet le symbole initial de la pile.

Pile	Entrée	Action
ϵ	$id := num; print(num)\$$	*shift*
id	$:= \ldots$	*shift*
$id :=$	$num \ldots$	*shift*
$id := num$	$; \ldots$	*reduce* $E \to num$
$id := E$	$; \ldots$	*reduce* $S \to id := E$
S	$; \ldots$	*shift*
$S;$	$print \ldots$	*shift*
$S; print$	$(\ldots$	*shift*
$S; print($	$num \ldots$	*shift*
$S; print(num$	$) \ldots$	*reduce* $E \to num$
$S; print(E$	$) \ldots$	*reduce* $L \to E$
$S; print(L$	$) \ldots$	*shift*
$S; print(L)$	$\$$	*reduce* $S \to print(L)$
$S; S$	$\$$	*reduce* $S \to S; S$
S	$\$$	*shift*
$S\$$	ϵ	*reduce* $S' \to S\$$
S'	ϵ	*accept*

Tab. 1 – Exemple d'analyse LR

5.1 Problèmes

Un certain nombre de questions doivent être clarifiées :

– La taille de la pile de l'APD peut évoluer suite à des opérations de *shift* et de *reduce*. On a besoin d'exprimer de façon synthétique ce qui se trouve sur la pile.
– Un membre droit d'une règle peut être facteur droit d'une autre règle comme dans :

$$A \to \alpha_1\alpha_2 \quad B \to \alpha_2$$

Si $\alpha_1\alpha_2$ sont sur la pile, il faut décider quelle réduction appliquer. On appelle cette situation un *conflit reduce/reduce*.
– Si une partie droite d'une règle est sur la pile, faut-il exécuter un *reduce* ou un *shift* ? Une possibilité serait de privilégier toujours un *reduce*, mais dans certaines situations il peut être préférable d'exécuter un *shift* comme dans la deuxième ligne de la table 1.

5.2 Pragmatique

Le processus de développement d'un APD est automatisé grâce à des outils comme YACC.
– On commence par écrire une grammaire, le plus souvent en imitant une autre grammaire qui marche.
– YACC met en évidence un certain nombre de conflits *shift/reduce* ou *reduce/reduce*.
– On cherche à régler les conflits. YACC offre la possibilité de spécifier simplement la façon d'associer un opérateur et les priorités entre opérateurs.

5.3 Survol des résultats les plus importants

– Un langage L a la *propriété du préfixe* si

$$w \in L, w = w_1w_2, w_2 \neq \epsilon \Rightarrow w_1 \notin L$$

c'est-à-dire si un mot w appartient à L alors aucun préfixe propre de w est dans L.
– On peut toujours modifier un langage L pour qu'il ait la propriété du préfixe. On introduit un nouveau symbole terminal \$ et on considère $L' = \{w\$ \mid w \in L\}$. Il est aisé de vérifier que L' a la propriété du préfixe. De plus, les mots dans L' sont simplement les mots de L avec un marqueur au fond.
– On associe à une grammaire $LR(k)$ un APD dont les décisions peuvent dépendre des k symboles en entrée.
– Les grammaires $LR(0)$ génèrent exactement les langages qui ont la propriété du préfixe et qui sont acceptés par un APD.
– Si un langage est accepté par un APD alors on peut construire une grammaire $LR(1)$ qui le génère.
– Le processus de génération d'un APD à partir d'une grammaire $LR(1)$ peut être inefficace. Pour cette raison, on s'intéresse aussi à des classes de grammaires $SLR(1)$ et $LALR(1)$ qui sont un compromis entre l'efficacité des grammaires $LR(0)$ et la généralité des grammaires $LR(1)$.

5.4 Grammaires $LR(0)$

Soit $G = (V, \Sigma, S, R)$ une grammaire non-contextuelle. On écrit $\alpha \Rightarrow_D \beta$ pour signifier que la réécriture concerne le symbole non-terminal de α le plus à droite.

Définition 5.2 *Soit $\cdot$ un symbole tel que $\cdot \notin V$. On définit l'ensemble des items :*

$$Item = \{A \to \alpha \cdot \beta \mid A \to \alpha\beta\}$$

Définition 5.3 *Un mot $\gamma \in V^*$ est un* préfixe admissible *s'il existe une dérivation droite :*

$$S \Rightarrow_D^* \delta Aw \Rightarrow_D \delta\alpha\beta w$$

où $w \in \Sigma^*$, $\delta\alpha = \gamma$ et $A \to \alpha\beta$ est une règle. Dans ce cas, on dit aussi que l'item $A \to \alpha \cdot \beta$ est valide *pour γ.*

Exemple 5.4 *On considère la grammaire*

$$S' \to Sc \quad S \to SA \mid A \quad A \to aSb \mid ab$$

avec S' symbole initial. L'ensemble Item est composé par

$$\{S' \to \cdot Sc, S' \to S \cdot c, S' \to Sc\cdot, S \to \cdot SA, \ldots\}$$

On prend comme dérivation droite :

$$S' \Rightarrow_D Sc \Rightarrow_D SAc \Rightarrow_D SaSbc$$

Pour déterminer les préfixes admissibles par rapport à $SaSbc$ on cherche d'abord un sous-mot qui correspond à une partie droite d'une règle. Dans notre cas la seule possibilité est aSb. Par ailleurs, $S' \Rightarrow_D Sc \Rightarrow_D SAc \Rightarrow_D SaSbc$. Donc on peut prendre $w = c$ et on obtient comme préfixes admissibles : S, Sa, SaS et $SaSb$.

Construction de l'AFD Comment savoir si un préfixe est admissible ? Un résultat remarquable est qu'on peut construire un AFD qui accepte exactement les préfixes admissibles et qui au passage, nous donne aussi l'ensemble des items valides pour le préfixe.

On commence par construire un AFN (*automate fini non-déterministe*) M et on obtient l'AFD par une procédure standard de déterminisation.

- L'alphabet d'entrée de l'AFN est l'ensemble V de symboles terminaux et non terminaux de la grammaire.
- L'ensemble des états est donné par $Q = \{q_0\} \cup Item$. On a donc l'ensemble des items et un état q_0 qu'on prend comme état initial.
- Tous les états sauf q_0 sont des états finaux.
- La fonction de transition δ a le type :

$$\delta : Q \times (V \cup \{\epsilon\}) \to 2^Q$$

et elle est définie par les conditions suivantes :

1. $\delta(q_0, \epsilon) = \{S \to \cdot\alpha \mid S \to \alpha \in R\}$.
2. $\delta(A \to \alpha \cdot B\beta, \epsilon) = \{B \to \cdot\gamma \mid B \to \gamma \in R\}$.
3. $\delta(A \to \alpha \cdot X\beta, X) = \{A \to \alpha X \cdot \beta\}$ où $X \in V$.

Exemple 5.5 *On construit l'AFN associé à la grammaire dans l'exemple 5.4. Soient :*

$0 = q_0$	$1 = S' \to \cdot Sc$	$2 = S' \to S \cdot c$	$3 = S' \to Sc\cdot$	$4 = S \to \cdot SA$	$5 = S \to \cdot A$
$6 = S \to A\cdot$	$7 = S \to S \cdot A$	$8 = A \to \cdot aSb$	$9 = A \to \cdot ab$	$10 = S \to SA\cdot$	$11 = A \to a \cdot Sb$
$12 = A \to a \cdot b$	$13 = A \to aS \cdot b$	$14 = A \to ab\cdot$	$15 = A \to aSb\cdot$		

Les transitions sont définies par le tableau suivant :[1]

$\to$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		ϵ														
1			S		ϵ	ϵ										
2				c												
3																
4					ϵ	ϵ		S								
5							A		ϵ	ϵ						
6																
7									ϵ	ϵ	A					
8												a				
9													a			
10																
11					ϵ	ϵ								S		
12															b	
13																b
14																
15																

Une configuration de l'AFN est un couple (q, w) où q est un état et w est un mot. On écrit $(q, w) \vdash (q', w')$ ($(q, w) \vdash^* (q', w')$) si l'AFN peut aller de la première à la deuxième configuration avec un pas de calcul (un nombre fini, éventuellement 0, de pas de calcul).

Théorème 5.6 $(q_0, \gamma) \vdash^* (A \to \alpha \cdot \beta, \epsilon)$ *ssi* γ *est un préfixe admissible et* $A \to \alpha \cdot \beta$ *est valide pour* γ.

Idée de la preuve. ($\Rightarrow$) On procède par récurrence sur la longueur du chemin le plus court étiqueté avec γ qui va de l'état initial q_0 à un item (état) $A \to \alpha \cdot \beta$.

- Si le chemin a longueur 1 alors $\gamma = \epsilon$, l'item a la forme $S \to \cdot\beta$, ϵ est un préfixe admissible pour la dérivation droite $S \Rightarrow_D \beta$ et $S \to \cdot\beta$ est valide pour ϵ.
- Si la dernière arête est étiquetée par $X \in V$ alors l'arête va d'un item $A \to \alpha' \cdot X\beta$ à un item $A \to \alpha' X \cdot \beta$ avec $\alpha = \alpha' X$. Par hypothèse de récurrence,

$$S \Rightarrow_D^* \delta A w \Rightarrow_D \delta\alpha' X\beta w$$

  $\gamma' = \delta\alpha'$ et $A \to \alpha' \cdot X\beta$ est valide pour γ'. Mais alors la même réduction montre que $\gamma = \gamma' X$ est admissible et que $A \to \alpha' X \cdot \beta$ est valide pour γ.
- Si la dernière arête est étiquetée par ϵ alors l'arête va d'un item $B \to \alpha_1 \cdot A\beta_1$ à un item $A \to \cdot\beta$. Par hypothèse de récurrence, $B \to \alpha_1 \cdot A\beta_1$ est valide pour γ. Donc

$$S \Rightarrow_D^* \delta B w \Rightarrow_D \delta\alpha_1 A\beta_1 w$$

  et $\gamma = \delta\alpha_1$. Si tous les symboles de la grammaire sont utiles, on peut supposer que $\beta_1 \Rightarrow_D^* w_1$. Donc

$$S \Rightarrow_D^* \delta B w \Rightarrow_D \delta\alpha_1 A\beta_1 w \Rightarrow_D^* \delta\alpha_1 A w_1 w$$

  avec $\gamma = \delta\alpha_1$ admissible et $A \to \cdot\beta$ valide pour γ.

[1] Il faut lire le tableau de la façon suivante : si le symbole X a ordonnée q et abscisse q' alors l'automate peut faire une transition X de q à q'.

($\Leftarrow$) On suppose

$$S \Rightarrow_D^* \gamma_1 A w \Rightarrow_D \gamma_1 \alpha \beta w$$

où $A \to \alpha\beta$ est un règle et $\gamma = \gamma_1\alpha$. On montre par récurrence sur la longueur de la dérivation droite que

$$(q_0, \gamma_1) \vdash^* (A \to \cdot\alpha\beta, \epsilon)$$

Il suit, par construction de l'automate, que $(A \to \cdot\alpha\beta, \alpha) \vdash^* (A \to \alpha \cdot \beta, \epsilon)$ et donc que $(q_0, \gamma) \vdash^* (A \to \alpha \cdot \beta, \epsilon)$.

- Le cas de base est immédiat : si $S \Rightarrow_D \beta$ alors $(q_0, \epsilon) \vdash (S \to \cdot\beta, \epsilon)$.
- Pour le pas inductif, supposons que

  $$S \Rightarrow_D^* \gamma_2 B w_1 \Rightarrow_D \gamma_2\gamma_3 A \gamma_4 w_1 \Rightarrow_D^* \gamma_2\gamma_3 A w_2 w_1$$

  où $\gamma_1 = \gamma_2\gamma_3$ et $w = w_2w_1$. Par hypothèse de récurrence : $(q_0, \gamma_2) \vdash^* (B \to \cdot\gamma_3 A\gamma_4, \epsilon)$. On dérive : $(B \to \cdot\gamma_3 A\gamma_4, \gamma_3) \vdash^* (B \to \gamma_3 \cdot A\gamma_4, \epsilon) \vdash (A \to .\alpha\beta, \epsilon)$. Donc : $(q_0, \gamma_1) \vdash^* (A \to \cdot\alpha\beta, \epsilon)$. •

A partir de l'AFN on dérive par déterminisation un AFD dont les états sont des ensembles d'items.

Définition 5.7 *Un item est* complet *s'il a la forme $A \to \alpha\cdot$, c'est-à-dire le symbole spécial $\cdot$ est le symbole le plus à droite.*

Définition 5.8 *La grammaire G est LR(0) si*

1. *Le symbole initial ne paraît pas à droite d'une règle (une condition technique qui n'est pas restrictive en pratique).*
2. *Si un état de l'AFD associé contient un item complet alors il ne contient pas d'autres items.*

Exemple 5.9 *On construit l'AFD associé à la grammaire dans l'exemple 5.4. Soient :*

$0 = \{0, 1, 4, 5, 8, 9\}$ $1 = \{2, 7, 8, 9\}$ $2 = \{6\}$ $3 = \{4, 5, 8, 9, 11, 12\}$ $4 = \{3\}$
$5 = \{10\}$ $6 = \{7, 8, 9, 13\}$ $7 = \{14\}$ $8 = \{15\}$

Les transitions sont :

$\to$	0	1	2	3	4	5	6	7	8
0		S	A	a					
1				a	c	A			
2									
3			A	a			S	b	
4									
5									
6				a					b
7									
8									

On peut vérifier que la grammaire est LR(0).

Construction de l'APD On construit un APD qui accepte le langage généré par une grammaire $LR(0)$.

- L'APD garde sur la pile un *préfixe admissible* $\gamma = X_1 \cdots X_k$. Plus précisément, on intercale les symboles dans Γ avec les états de l'AFD de façon à avoir $q_0 X_1 q_1 \cdots X_k q_k$ avec $q_{i+1} = \delta(q_i, X_{i+1})$, $i = 0, \ldots, k-1$.
- Si l'état au sommet de la pile est un ensemble composé d'un item complet, par exemple $q_k = \{A \to X_i \cdots X_k \cdot\}$, alors on effectue un *reduce*. La nouvelle pile a la forme

$$q_0 X_1 q_1 \cdots X_{i-1} q_{i-1} A q$$

  où $q = \delta(q_{i-1}, A)$.
- Si q_k n'est pas complet on effectue un *shift*. Si a est le symbole lu, la nouvelle pile est $q_0 X_1 q_1 \cdots X_k q_k a q$ si $q = \delta(q_k, a)$.
- On accepte quand S symbole initial est sur la pile (le fait de faire un *reduce* sur le symbole initial signifie que la dérivation est terminée car le symbole initial ne peut pas paraître à droite d'une règle).

Exemple 5.10 *On simule le comportement de l'APD associé à la grammaire de l'exemple 5.4 sur l'entrée aababbc.*

Pile	*Entrée*	*Action*
0	*aababbc*	*shift*
0*a*3	*ababbc*	*shift*
0*a*3*a*3	*babbc*	*shift*
0*a*3*a*3*b*7	*abbc*	*reduce* $A \to ab$
0*a*3*A*2	*abbc*	*reduce* $S \to A$
0*a*3*S*6	*abbc*	*shift*
0*a*3*S*6*a*3	*bbc*	*shift*
0*a*3*S*6*a*3*b*7	*bc*	*reduce* $A \to ab$
0*a*3*S*6*A*5	*bc*	*reduce* $S \to SA$
0*a*3*S*6	*bc*	*shift*
0*a*3*S*6*b*8	*c*	*reduce* $A \to aSb$
0*A*2	*c*	*reduce* $S \to A$
0*S*1	*c*	*shift*
0*S*1*c*4	_	*reduce* $S' \to Sc$
0S'	_	*accept*

Exercice 5.11 *On considère la grammaire suivante avec symbole initial S :*

$$S \to A\$ \qquad A \to BA \mid \epsilon \qquad B \to aB \mid b$$

Expliquez pourquoi cette grammaire n'est pas LR(0).
Suggestion $A \to \cdot$ *est un item complet.*

Exercice 5.12 *On considère la grammaire G suivante :*

$$S \to C, \quad C \to 0 \mid aCb \ .$$

1. *Donnez la représentation graphique de l'automate fini non-déterministe qui reconnaît les préfixes admissibles.*
2. *La grammaire est-elle LR(0) ? Expliquez.*

3. *Dérivez de l'analyse un automate à pile déterministe qui accepte le langage généré par la grammaire.*

Exercice 5.13 *On considère (à nouveau) la grammaire :*

$$S \to A \mid B \quad A \to aAb \mid 0 \quad B \to aBbb \mid 1$$

Vérifiez que la grammaire est LR(0) et dérivez l'automate à pile déterministe qui accepte le langage généré par la grammaire.

Exercice 5.14 *On considère la grammaire suivante avec S symbole initial :*

$$\begin{array}{ll} S \to E\$ & E \to id \\ E \to id(E) & E \to E + id \end{array}$$

1. *Donnez la représentation graphique d'un automate fini non-déterministe qui reconnaît les préfixes admissibles de la grammaire.*
2. *La grammaire est-elle LR(0) ? Justifiez votre réponse.*

5.5 Grammaires LR(1)

Les principes de construction développés pour les grammaires LR(0) se généralisent aux grammaires LR(1). Soient $G = (V, \Sigma, S, R)$ une grammaire algébrique et \$ un nouveau symbole qui servira à marquer la fin du mot à générer.

Item Un *item* est maintenant un couple $(A \to \alpha \cdot \beta, T)$ où $A \to \alpha\beta \in R$ est une règle de la grammaire et T est un sous-ensemble de $\Sigma \cup \{\$\}$.

AFN On construit un AFN dont les états Q sont les items plus un état initial q_o. La fonction de transition δ a le type :

$$\delta : Q \times (V \cup \{\epsilon\}) \to 2^Q$$

et elle est définie par les conditions suivantes :

1. $\delta(q_0, \epsilon) = \{(S \to \cdot\alpha, \{\$\}) \mid S \to \alpha \in R\}$.
2. $\delta((A \to \alpha \cdot B\beta, T), \epsilon) = \{(B \to \cdot\gamma, T_\beta) \mid B \to \gamma \in R\}$ où

$$T_\beta = \begin{cases} \mathit{First}(\beta) \cup T & \text{si } \mathit{null}(\beta) \\ \mathit{First}(\beta) & \text{autrement} \end{cases}$$

3. $\delta((A \to \alpha \cdot X\beta, T), X) = \{(A \to \alpha X \cdot \beta, T)\}$ où $X \in V$.

AFD On déterminise l'AFN pour obtenir un AFD équivalent. On dit que la grammaire est LR(1) si : (i) le symbole initial ne paraît pas à droite d'une règle et (ii) si un état de l'AFD associé contient un item complet $(A \to \alpha\cdot, T)$ alors :

1. Si l'état contient un item $(B \to \alpha' \cdot a'\beta', T')$ alors $a' \notin T$.
2. Si l'état contient un item $(B \to \alpha'\cdot, T')$ alors $T \cap T' = \emptyset$.

Intuitivement, la condition 1 évite les conflits *reduce-shift* et la condition 2 évite les conflits *reduce-reduce*.

APD Si la grammaire est LR(1) alors on peut construire un APD qui reconnaît le langage généré par la grammaire et qui effectue des opérations de *shift* et de *reduce*. La nouveauté est que maintenant l'APD peut prendre en compte le prochain symbole en entrée.[2] Comme dans le cas LR(0), la pile a la forme $q_0X_1q_1\cdots X_kq_k$ où $q_{i+1} = \delta(q_i, X_{i+1})$, $i = 0, \ldots, k-1$ et δ est la fonction de transition de l'AFD. L'APD effectue trois types d'actions.

reduce Si l'état au sommet de la pile contient un item complet $(A \to X_i \cdots X_k\cdot, T)$ avec $A \neq S$ et si le prochain symbole en entrée appartient à T alors on effectue une action *reduce* et la nouvelle pile a la forme $q_0X_1q_1 \cdots X_{i-1}q_{i-1}Aq$ où $q = \delta(q_{i-1}, A)$.

shift Si l'état au sommet de la pile contient un item $(A \to \alpha \cdot aB, T)$ et a est le prochain symbole en entrée alors on effectue une action *shift* et la nouvelle pile est $q_0X_1q_1 \cdots X_kq_kaq$ où $q = \delta(q_k, a)$.

accept Si l'état au sommet de la pile contient un item $(S \to X_1 \cdots X_k\cdot, \{\$\})$ et si le prochain symbole en entrée est \$ alors on *accepte*.

Les conditions qui définissent une grammaire LR(1) assurent qu'à chaque moment du calcul au plus une action sera possible.

Exemple 5.15 *On considère la grammaire G :*

$$S \to A \qquad A \to BA \mid \epsilon \qquad B \to aB \mid b$$

Cette grammaire génère le langage régulier $(a^*b)^*$. Ce langage n'a pas la propriété du préfixe et donc la grammaire G ne peut pas être LR(0). On pourrait considérer la grammaire G' :

$$S \to A\$ \qquad A \to BA \mid \epsilon \qquad B \to aB \mid b$$

mais il est facile de vérifier que la grammaire obtenue n'est toujours pas LR(0).

On calcule les fonctions null, First et Follow.

	null	*First*	*Follow*
S	*oui*	$\{a, b\}$	$\emptyset$
A	*oui*	$\{a, b\}$	$\emptyset$
B	*non*	$\{a, b\}$	$\{a, b\}$

On peut vérifier au passage que la grammaire G est LL(1). Il s'agit donc d'un exemple de grammaire qui est LL(1) mais pas LR(0). On sait que les grammaires LL(1) et LR(0) sont incomparables *et qu'elles sont contenues strictement dans les grammaires LR(1).*

Nous allons vérifier que la grammaire G est LR(1). D'abord on construit l'AFN. Soient :

$0 = q_0$ $\quad 1 = (S \to \cdot A, \{\$\})$ $\quad 2 = (S \to A\cdot, \{\$\})$ $\quad 3 = (A \to \cdot BA, \{\$\})$
$4 = (A \to \cdot, \{\$\})$ $\quad 5 = (A \to B \cdot A, \{\$\})$ $\quad 6 = (B \to \cdot aB, \{a, b, \$\})$ $\quad 7 = (B \to \cdot b, \{a, b, \$\})$
$8 = (A \to BA\cdot, \{\$\})$ $\quad 9 = (B \to a \cdot B, \{a, b, \$\})$ $\quad 10 = (B \to b\cdot, \{a, b, \$\})$ $\quad 11 = (B \to aB\cdot, \{a, b, \$\})$

Les transitions sont définies par le tableau suivant :

[2] La technique utilisée pour se ramener à un APD standard consiste à ajouter le symbole spécial \$ à l'entrée et à 'mémoriser' le prochain symbole en entrée dans l'état de l'automate.

$\rightarrow$	0	1	2	3	4	5	6	7	8	9	10	11
0		ϵ										
1			A	ϵ	ϵ							
2												
3						B	ϵ	ϵ				
4												
5				ϵ	ϵ				A			
6										a		
7											b	
8												
9							ϵ	ϵ				B
10												
11												

On construit l'AFD associé. Soient :

$0 = \{0, 1, 3, 4, 6, 7\}$ $1 = \{2\}$ $2 = \{3, 4, 5, 6, 7\}$ $3 = \{6, 7, 9\}$ $4 = \{10\}$
$5 = \{8\}$ $6 = \{11\}$

Les transitions sont :

$\rightarrow$	0	1	2	3	4	5	6
0		A	B	a	b		
1							
2			B	a	b	A	
3				a	b		B
4							
5							
6							

L'état 3 *ne contient pas d'item complet. Les états* $1, 4, 5, 6$ *contiennent un seul item qui est complet. Les états* 0 *et* 2 *contiennent un seul item complet* $(A \rightarrow \cdot, \{\$\})$ *et dans les autres items,* \$ *ne paraît pas à droite de* $\cdot$*. Donc la grammaire* G *est LR(1).*

On termine l'exemple en simulant le comportement de l'APD associé à la grammaire sur l'entrée $aabb\$$*.*

Pile	*Entrée*	*Action*
0	$aabb\$$	*shift*
$0a3$	$abb\$$	*shift*
$0a3a3$	$bb\$$	*shift*
$0a3a3b4$	$b\$$	*reduce* $B \rightarrow b$
$0a3a3B6$	$b\$$	*reduce* $B \rightarrow aB$
$0a3B6$	$b\$$	*reduce* $B \rightarrow aB$
$0B2$	$b\$$	*shift*
$0B2b4$	\$	*reduce* $B \rightarrow b$
$0B2B2$	\$	*reduce* $A \rightarrow \epsilon$
$0B2B2A5$	\$	*reduce* $A \rightarrow BA$
$0B2A5$	\$	*reduce* $A \rightarrow BA$
$0A1$	\$	*reduce* $S \rightarrow A$
$0S$	\$	*accepte*

6 Évaluation et typage d'un langage impératif

On présente les règles d'évaluation et de typage d'un petit langage impératif. Par rapport au langage d'expressions considéré dans la section 2, on voit paraître plusieurs concepts nouveaux :

- la notion d'*environnement* qui suggère une mise-en-oeuvre efficace de la substitution.
- la notion de *mémoire* qui permet de modéliser l'affectation.
- la *gestion du contrôle* : les branchements, les boucles, le *goto*,...
- le *passage de paramètres* aux procédures.
- la *gestion de l'environnement* ou comment associer une valeur à un nom.

6.1 Syntaxe

Dans un souci de lisibilité, on utilise une syntaxe *concrète*, étant entendu que l'évaluateur opère sur l'arbre de syntaxe *abstraite* généré par l'analyseur syntaxique.[3]

Types Les types comprennent deux types de base *bool* et *int* ainsi que les types produit et référence :

$$\tau ::= bool \mid int \mid \tau * \tau \mid ref\ \tau$$

On dénote avec $(\tau_1, \ldots, \tau_n) \to \tau$ le type d'une opération qui reçoit n arguments de type $\tau_1, \ldots, \tau_n$ et rend un résultat de type τ. On dénote aussi avec $(\tau_1, \ldots, \tau_n)$ le type d'une procédure qui attend n paramètres de type $\tau_1, \ldots, \tau_n$ (et ne rend pas de résultat).

Valeurs Soit **Z** l'ensemble des entiers avec éléments $n, m, \ldots$ Soit *Loc* un ensemble infini de *locations* avec éléments $\ell, \ell', \ldots$. Concrètement on peut penser à une location comme à une adresse de mémoire. Les valeurs comprennent les valeurs de base, les couples de valeurs et les locations :

$$v ::= true \mid false \mid n \mid Pair(v, v) \mid \ell$$

Soit *Val* l'ensemble des valeurs (qui contient l'ensemble des locations).

Expressions La catégorie syntaxique des expressions comprend des opérateurs *op* sur les types de base *bool* et *int*. Par exemple : $+, *, \wedge, >, \ldots$ A chaque opérateur *op* on associe l'interprétation attendue $\underline{op}$. Par exemple, $\underline{+}$ est l'addition sur les entiers.

$$\begin{array}{lll} id & ::= & x \mid y \mid \ldots \\ e & ::= & v \mid id \mid op(e_1, \ldots, e_n) \mid Pair(e, e) \mid Fst(e) \mid Snd(e) \mid ref\ e \mid !e \end{array}$$

Les expressions comprennent aussi un constructeur de couple et les projections associées ainsi qu'un générateur de référence et l'opérateur de déréférenciation associé.

Déclarations Une déclaration est une liste d'associations entre identificateurs et expressions :

$$D ::= \epsilon \mid (let\ id = e); D$$

Commandes Les commandes comprennent une affectation et des opérateurs de base pour manipuler le flot du contrôle :

$$S ::= id := e \mid (while\ e\ do\ S) \mid (if\ e\ then\ S\ else\ S) \mid S; S$$

[3] Dans la suite on adopte la notation de *Backus-Naur* : pour spécifier une grammaire algébrique on écrit $S ::= \cdots$ plutôt que $S \to \cdots$

Programmes Un programme est constitué d'une déclaration suivie par une commande :

$$P ::= D; S$$

Exemple 6.1 *Voici un exemple de programme :*

$$\begin{array}{l} \mathit{let}\ x = \mathit{ref}\ !\mathit{input}; \\ \mathit{let}\ y = \mathit{ref}\ 1; \\ (\mathit{while}\ !x > 0\ \mathit{do}\ y := 2{*}!y;\ x :=!x - 1); \\ \mathit{output} :=!y \end{array}$$

Ici on suppose que certaines variables comme input et output sont prédéfinies. Un programme utilise ces variables pour lire l'entrée et écrire le résultat du calcul.

6.2 Évaluation

Pour évaluer expressions, déclarations, commandes et programmes nous avons besoin de deux paramètres : un environnement η et une mémoire μ. Soit Id l'ensemble des identificateurs. On définit :

- Un *environnement* $\eta : Id \rightharpoonup Val$ comme une fonction partielle à domaine fini de Id dans Val.
- Une *mémoire* $\mu : Loc \rightharpoonup Val$ comme une fonction partielle à domaine fini de Loc dans Val.

On utilise les *jugements* suivants :

$$(e, \eta, \mu) \Downarrow (v, \mu'), \qquad (D, \eta, \mu) \Downarrow (\eta', \mu'), \qquad (S, \eta, \mu) \Downarrow \mu', \qquad (P, \eta, \mu) \Downarrow \mu'\ .$$

On écrit $\ell = \mathit{new}(\eta, \mu)$ pour dire que ℓ est une location qui ne parait pas dans $\mathit{im}(\eta) \cup \mathit{dom}(\mu) \cup \mathit{im}(\mu)$. Cette opération peut toujours être effectuée car domaine et image sont des ensembles finis et on suppose que Loc est un ensemble infini.

Expressions

$$\frac{}{(v, \eta, \mu) \Downarrow (v, \mu)} \qquad \frac{\eta(x) \text{ défini}}{(x, \eta, \mu) \Downarrow (\eta(x), \mu)}$$

$$\frac{(e, \eta, \mu) \Downarrow (v, \mu') \quad \ell = \mathit{new}(\eta, \mu')}{(\mathit{ref}\ e, \eta, \mu) \Downarrow (\ell, \mu'[v/\ell])} \qquad \frac{(e, \eta, \mu) \Downarrow (\ell, \mu')}{(!e, \eta, \mu) \Downarrow (\mu'(\ell), \mu')}$$

$$\frac{\begin{array}{c}(e_1, \eta, \mu) \Downarrow (v_1, \mu_1) \cdots (e_n, \eta, \mu_{n-1}) \Downarrow (v_n, \mu_n) \\ v_1, \ldots, v_n \text{ dans le domaine de définition de } \mathit{op}\end{array}}{(\mathit{op}(e_1, \cdots, e_n), \eta, \mu) \Downarrow (\underline{\mathit{op}}(v_1, \ldots, v_n), \mu_n)}$$

$$\frac{(e_1, \eta, \mu) \Downarrow (v_1, \mu_1) \quad (e_2, \eta, \mu_1) \Downarrow (v_2, \mu_2)}{(\mathit{Pair}(e_1, e_2), \eta, \mu) \Downarrow (\mathit{Pair}(v_1, v_2), \mu_2)}$$

$$\frac{(e, \eta, \mu) \Downarrow (\mathit{Pair}(v_1, v_2), \mu')}{(\mathit{Fst}(e), \eta, \mu) \Downarrow (v_1, \mu')} \qquad \frac{(e, \eta, \mu) \Downarrow (\mathit{Pair}(v_1, v_2), \mu')}{(\mathit{Snd}(e), \eta, \mu) \Downarrow (v_2, \mu')}$$

Exemple 6.2 *On suppose* $\eta_0(x) = \ell_x$ *et* $\mu_0(\ell_x) = \mathit{true}$*. On dérive :*

$$(\mathit{Snd}(!(\mathit{ref}\ \ (\mathit{Pair}(3, !x)))), \eta_0, \mu_0) \Downarrow (\mathit{true}, \mu_0[\mathit{Pair}(3, \mathit{true})/\ell_1])$$

Déclarations

$$\frac{}{(\_, \eta, \mu) \Downarrow (\eta, \mu)} \qquad \frac{(e, \eta, \mu) \Downarrow (v, \mu') \quad (D, \eta[v/x], \mu') \Downarrow (\eta'', \mu'')}{(let\ x = e; D, \eta, \mu) \Downarrow (\eta'', \mu'')}$$

Exercice 6.3 *Évaluez la liste de déclarations suivante :*

$$let\ x = ref\ 3;$$
$$let\ y = !x + 3 :$$

Remarque 6.4 *Il est intéressant de comparer le traitement du let ci-dessus avec celui décrit en section 2.5 pour le simple langage d'expressions. Dans le langage d'expressions on remplace, par une opération de substitution, l'identificateur par une valeur. Cette opération de substitution n'est pas triviale et elle peut s'avérer coûteuse. Dans la pratique de la mise-en-oeuvre, il est préférable d'introduire une notion d'*environnement *comme nous l'avons fait pour le langage impératif et d'enregistrer dans l'environnement l'association entre identificateur et valeur. Bien sûr on pourrait appliquer cette idée aussi au langage d'expressions de la section 2.5. Dans ce cas le jugement deviendrait* $(b, \eta) \Downarrow (v, \eta')$ *et la règle pour le let s'écrirait comme suit :*

$$\frac{(e', \eta) \Downarrow (v', \eta') \quad (b, \eta[v'/x]) \Downarrow (v, \eta'')}{(let\ x = e'\ in\ b, \eta) \Downarrow (v, \eta'')}$$

Commandes

$$\frac{\eta(x) = \ell \quad (e, \eta, \mu) \Downarrow (v, \mu')}{(x := e, \eta, \mu) \Downarrow \mu'[v/\ell]} \qquad \frac{(S_1, \eta, \mu) \Downarrow \mu_1 \quad (S_2, \eta, \mu_1) \Downarrow \mu_2}{(S_1; S_2, \eta, \mu) \Downarrow \mu_2}$$

$$\frac{(e, \eta, \mu) \Downarrow (false, \mu')}{(while\ e\ do\ S, \eta, \mu) \Downarrow \mu'} \qquad \frac{(e, \eta, \mu) \Downarrow (true, \mu') \quad (S; (while\ e\ do\ S), \eta, \mu') \Downarrow \mu''}{(while\ e\ do\ S, \eta, \mu) \Downarrow \mu''}$$

$$\frac{(e, \eta, \mu) \Downarrow (true, \mu') \quad (S_1, \eta, \mu') \Downarrow \mu''}{(if\ e\ then\ S_1\ else\ S_2, \eta, \mu) \Downarrow \mu''} \qquad \frac{(e, \eta, \mu) \Downarrow (false, \mu') \quad (S_2, \eta, \mu') \Downarrow \mu''}{(if\ e\ then\ S_1\ else\ S_2, \eta, \mu) \Downarrow \mu''}$$

Exercice 6.5 *Évaluez la commande suivante dans un environnement* η *tel que* $\eta(x) = \ell_x$.

$$x := 1;$$
$$while\ !x > 0\ do$$
$$x :=!x - 1$$

Programmes

$$\frac{(D, \eta, \mu) \Downarrow (\eta', \mu') \quad (S, \eta', \mu') \Downarrow \mu''}{(D; S, \eta, \mu) \Downarrow \mu''}$$

Remarque 6.6 *On notera que :*

- *L'évaluation d'une expression n'affecte pas l'environnement et peut éventuellement produire une extension de la mémoire.*
- *L'évaluation d'une déclaration peut modifier l'environnement et peut provoquer l'évaluation d'une expression.*

– *L'évaluation d'une commande peut modifier la mémoire et peut provoquer l'évaluation d'une expression.*

Exercice 6.7 *Évaluez le programme de l'exemple 6.1 dans l'environnement η_0 et la mémoire μ_0 tels que $\eta_0(input) = \ell_0$ et $\mu(\ell_0) = 1$.*

Exercice 6.8 *On modifie la catégorie syntaxique des commandes S en remplaçant la commande while par la commande goto :*

$$S ::= id := e \mid lab : S \mid goto\ lab \mid (if\ e\ then\ S) \mid S; S$$

Ici lab est une nouvelle catégorie syntaxique d'étiquettes qu'on dénote par $a, b, c, \ldots$

Pour décrire l'évaluation de ces nouvelles commandes on élargit le domaine de définition des environnements aux étiquettes. Si a est une étiquette, η est un environnement et $\eta(a)$ est défini alors $\eta(a)$ est une commande. Les règles d'évaluation pour les deux nouvelles commandes introduites sont alors les suivantes :

$$\frac{(S, \eta[S/a], \mu) \Downarrow \mu'}{(a : S, \eta, \mu) \Downarrow \mu'} \qquad \frac{(\eta(a), \eta, \mu) \Downarrow \mu'}{(goto\ a, \eta, \mu) \Downarrow \mu'}$$

1. *Complétez la description de l'évaluation en donnant les règles pour l'affectation $x := e$, le branchement $(if\ e\ then\ S)$ et la séquentialisation $S_1; S_2$.*
2. *Proposez un schéma de traduction de la commande $(while\ e\ do\ S)$ dans le langage avec goto.*
3. *Appliquez votre schéma de traduction à la commande :*

$$\begin{array}{l}(while\ x\ do\\ \quad (while\ y\ do\ S_1))\ ;\\ (while\ z\ do\ S_2)\end{array}$$

6.3 Extension avec procédures

On considère maintenant une extension du langage avec procédures.

Syntaxe révisée

Déclarations de procédure On ajoute les déclarations de procédure.

$$DP ::= \epsilon \mid (procedure\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) = D; S)\ ;\ DP$$

Une procédure se compose d'un nom, d'une liste de paramètres formels typés et d'un corps de procédure $D; S$.

Commandes On ajoute l'appel de procédure à la liste des commandes.

$$S ::= \cdots \mid f(e_1, \ldots, e_n)$$

Programmes Maintenant, un programme se compose d'une liste éventuellement vide de déclarations de procédure et d'un corps principal $D; S$.

$$P ::= DP; D; S$$

Évaluation révisée On définit *Pid* comme l'ensemble des identificateurs de procédures. On définit *VProc* comme l'ensemble des couples de la forme $(x_1 \cdots x_n, D; S)$, c'est-à-dire une liste d'identificateurs et le 'code' d'un corps de procédure. Un environnement $\eta : Id \cup Pid \rightharpoonup Val \cup VProc$ est maintenant une fonction partielle à domaine fini qui associe une valeur aux identificateurs et un élément de *VProc* aux identificateurs de procédure. Si η est un environnement, on dénote par $\eta_{|Pid}$ la restriction de η à *Pid*. Il faut ajouter des règles pour la déclaration de procédure et pour l'appel de procédure.

Déclaration de procédure On introduit un jugement $(DP, \eta) \Downarrow \eta$ qui est défini par les règles :

$$\frac{}{(\_, \eta) \Downarrow \eta} \qquad \frac{(DP, \eta[(\mathbf{x},\ D; S)/f]) \Downarrow \eta'}{((\mathit{procedure}\ f(\mathbf{x} : \vec{\tau}) = D; S); DP, \eta) \Downarrow \eta'}$$

L'effet d'une déclaration de procédure est d'associer au nom de la procédure ses paramètres formels et son code.

On remarquera que dans des langages plus généraux, cette information n'est pas suffisante. Dans ces langages on a besoin de connaître aussi l'environnement dans lequel la procédure doit être évaluée. On arrive ainsi à la notion de *clôture* qui est un couple (code, environnement). Cette notion sera développée dans la suite du cours.

Appel de procédure Les valeurs qui correspondent aux paramètres actuels de la procédure peuvent être des locations. Une procédure a donc la possibilité de retourner un résultat en modifiant la valeur contenue dans une location qui lui est transmise en paramètre. Au moment de l'appel de procédure il faut déterminer l'environnement dans lequel la procédure est appelée. Dans notre cas cet environnement comprend uniquement les noms des procédures.

$$\frac{\begin{array}{c}\eta(f) = (x_1 \cdots x_n, D; S) \\ (e_1, \eta, \mu) \Downarrow (v_1, \mu_1) \cdots (e_n, \eta, \mu_{n-1}) \Downarrow (v_n, \mu_n) \\ (D; S, \eta_{|Pid}[v_1/x_1, \ldots, v_n/x_n], \mu_n) \Downarrow \mu'\end{array}}{(f(e_1, \ldots, e_n), \eta, \mu) \Downarrow \mu'}$$

Programme La commande principale S est évaluée dans un environnement où toutes les procédures et les variables déclarées dans DP et D sont définies.

$$\frac{(DP, \eta) \Downarrow \eta' \quad (D; S, \eta', \mu) \Downarrow \mu'}{(DP; D; S, \eta, \mu) \Downarrow \mu'}$$

Remarque 6.9 *Il est important que dans l'évaluation du corps d'une procédure on cache l'environnement de l'appelant. Par exemple, dans*

$$\begin{array}{l}\mathit{procedure}\ f() = x :=\!!x + 1; \\ \mathit{let}\ x = \mathit{ref}\ 1; \\ f()\end{array}$$

la variable x ne doit pas être visible dans le corps de f et donc l'appel de f devrait produire une erreur au moment de l'exécution.

Exercice 6.10 *Évaluez :*

$$
\begin{array}{l}
(\mathit{procedure}\ f\ (x : \mathit{int}, y : \mathit{ref}\,\mathit{int}) = \\
\mathit{let}\ i = \mathit{ref}\ x; \\
y := 1;\ (\mathit{while}\ !i > 0\ \mathit{do}\ (y :=\!!i{*}!y; i :=\!!i - 1))); \\
\mathit{let}\ \mathit{out} = \mathit{ref}\ 0;\ \mathit{let}\ \mathit{in} = 2; \\
f(\mathit{in}, \mathit{out})
\end{array}
$$

6.4 Mise-en-oeuvre

On peut ajouter des règles pour les situations anormales. On donne une évaluation à erreur toutes le fois qu'on rencontre une des situations suivantes :

- on évalue une variable qui n'est pas dans l'environnement.
- les arguments de *op* ont le mauvais type.
- l'argument de *Fst* ou *Snd* n'est pas une paire.
- on dé-référence une valeur qui n'est pas une location.
- on cherche la valeur associée à une location qui n'est pas définie en mémoire.
- on affecte une variable qui n'est pas définie dans l'environnement ou qui est définie mais dont la valeur n'est pas une location.
- on a une condition de test pour *while* ou *if_then_else* qui n'est pas un booléen.
- on appelle une procédure qui n'est pas définie dans l'environnement ou on appelle une procédure avec le mauvais nombre d'arguments.

Par ailleurs, si une des évaluations en hypothèse donne erreur alors la conclusion donne erreur.

Exemple 6.11 *On introduit un symbole spécial err et on admet des jugements de la forme :*

$$(e, \eta, \mu) \Downarrow \mathit{err}, \quad (DP, \eta) \Downarrow \mathit{err}, \quad (D, \eta, \mu) \Downarrow \mathit{err}, \quad (S, \eta, \mu) \Downarrow \mathit{err}, \quad (P, \eta, \mu) \Downarrow \mathit{err}\ .$$

Si, par exemple, on considère la commande while on peut ajouter les règles suivantes :

$$\frac{(e, \eta, \mu) \Downarrow (v, \mu') \qquad v \notin \{\mathit{false}, \mathit{true}\}}{(\mathit{while}\ e\ \mathit{do}\ S, \eta, \mu) \Downarrow \mathit{err}} \qquad \frac{(e, \eta, \mu) \Downarrow \mathit{err}}{(\mathit{while}\ e\ \mathit{do}\ S, \eta, \mu) \Downarrow \mathit{err}}$$

$$\frac{(e, \eta, \mu) \Downarrow (\mathit{true}, \mu') \quad (S; (\mathit{while}\ e\ \mathit{do}\ S), \eta, \mu') \Downarrow \mathit{err}}{(\mathit{while}\ e\ \mathit{do}\ S, \eta, \mu) \Downarrow \mathit{err}}$$

Exercice 6.12 *Écrire les règles qui traitent les erreurs pour d'autres constructions du langage (expressions, déclarations de procédure, déclarations, commandes et programmes).*

Remarque 6.13 *On pourrait considérer d'autres types d'erreurs. Par exemple on peut associer une taille à chaque valeur.*

$$
\begin{array}{l}
|\mathit{true}| = |\mathit{false}| = |n| = |\ell| = 1 \\
|\mathit{Pair}(v_1, v_2)| = 1 + |v_1| + |v_2|
\end{array}
$$

L'espace alloué pour mémoriser une valeur serait proportionnel à sa taille. On pourrait donner une erreur si on alloue une valeur dont la taille excède celle de la mémoire allouée. Par exemple

$$
\begin{array}{l}
\mathit{let}\ x = \mathit{ref}\ 1; \\
x := \mathit{Pair}(!x, !x)
\end{array}
$$

pourrait produire une erreur.

La mise en oeuvre d'un évaluateur dans un langage qui permet les appels récursifs est très directe. Il s'agit de fixer des structures de données pour la représentation de l'environnement et de la mémoire et des opérations associées :

- générer une nouvelle location.
- créer des nouvelles associations dans l'environnement ou en mémoire.
- chercher une association dans l'environnement ou en mémoire.
- mettre à jour une association dans l'environnement ou en mémoire.

Ensuite il suffit de définir une fonction récursive 'Eval' qui est dirigée par la syntaxe (abstraite).

6.5 Liaison et évaluation

Le langage impératif considéré repose sur la *liaison statique* et l'*appel par valeur*. On analyse des variations possibles dans le cadre d'un simple langage d'expressions

$$e ::= \bot \mid n \mid Id \mid let\ Id = e\ in\ e \mid quote(e) \mid unquote(e)$$

où $\bot$ représente un calcul qui diverge, n est un entier et $Id = \{x, y, \ldots\}$ est l'ensemble des identificateurs. L'opérateur *quote* permet de bloquer l'évaluation d'une expression et l'opérateur *unquote* permet de la débloquer. Soit *Exp* l'ensemble des expressions. L'ensemble des valeurs v est défini par

$$v ::= n \mid quote(e)$$

On considère différentes stratégies d'évaluation : avec liaison dynamique ou statique et avec appel par nom ou par valeur.

Liaison dynamique Un environnement η est une fonction partielle de *Id* dans *Exp*. Soit *Env* l'ensemble des environnements. La relation d'évaluation $\Downarrow$ associe une valeur v à un couple $(e, \eta) \in Exp \times Env$. La relation est définie par les règles :

$$\frac{}{(v, \eta) \Downarrow v} \qquad \frac{(\eta(x), \eta) \Downarrow v}{(x, \eta) \Downarrow v}$$

$$\frac{(e, \eta) \Downarrow quote(e') \quad (e', \eta) \Downarrow v}{(unquote(e), \eta) \Downarrow v} \qquad \frac{(e, \eta) \Downarrow n}{(unquote(e), \eta) \Downarrow n}$$

et par la règle :

$$\text{(par nom)} \quad \frac{(e, \eta[e'/x]) \Downarrow v,}{(let\ x = e'\ in\ e, \eta) \Downarrow v} \quad \text{(par valeur)} \quad \frac{(e', \eta) \Downarrow u \quad (e, \eta[u/x]) \Downarrow v}{(let\ x = e'\ in\ e, \eta) \Downarrow v}$$

Liaison statique Dans ce cas l'ensemble des environnements *Env* est le plus petit ensemble de fonctions partielles (sur *Id*) tel que la fonction à domaine vide est un environnement et si $x_1, \ldots, x_n \in Id$, $e_1, \ldots, e_n \in Exp$ et $\eta_1, \ldots, \eta_n \in Env$ alors la fonction partielle telle que $\eta(x_i) = (e_i, \eta_i)$ pour $i = 1, \ldots, n$ est un environnement. La relation d'évaluation $\Downarrow$ associe à un couple $(e, \eta) \in Exp \times Env$, un couple (v, η'). On appelle aussi clôture un couple (e, η) constitué d'un code (une expression dans notre cas) et d'un environnement.

La relation d'évaluation est définie par les règles :

$$\frac{}{(v,\eta) \Downarrow (v,\eta)} \qquad \frac{\eta(x) \Downarrow (v,\eta')}{(x,\eta) \Downarrow (v,\eta')}$$

$$\frac{(e,\eta) \Downarrow (quote(e'),\eta') \quad (e',\eta') \Downarrow (v,\eta'')}{(unquote(e),\eta) \Downarrow (v,\eta'')} \qquad \frac{(e,\eta) \Downarrow (n,\eta')}{(unquote(e),\eta) \Downarrow (n,\eta')}$$

et par la règle :

$$\text{(par nom)} \quad \frac{(e,\eta[(e',\eta)/x]) \Downarrow (v,\eta_1)}{(let\ x = e'\ in\ e,\eta) \Downarrow (v,\eta_1)} \quad \text{(par valeur)} \quad \frac{(e',\eta) \Downarrow (u,\eta_2) \quad (e,\eta[(u,\eta_2)/x]) \Downarrow (v,\eta_1)}{(let\ x = e'\ in\ e,\eta) \Downarrow (v,\eta_1)}$$

Séparation On donne des exemples qui distinguent les différentes stratégies de liaison et d'évaluation.

- $let\ x = \bot\ in\ 3$ distingue appel par nom et appel par valeur dans les deux types de liaison. A savoir, l'évaluation converge par nom et diverge par valeur.
- Reste à comparer (1) dynamique+nom et statique+nom et (2) dynamique+valeur et statique+valeur. Soit :

  $$e_1 \equiv let\ x = 3\ in\ e_2, \quad e_2 \equiv let\ y = x\ in\ e_3, \quad e_3 \equiv let\ x = 5\ in\ y\ .$$

  En dynamique par nom, $(e_1,\emptyset) \Downarrow 5$. En dynamique par valeur, $(e_1,\emptyset) \Downarrow 3$. En statique par nom $(e_1,\emptyset) \Downarrow (3,\emptyset)$. En statique par valeur $(e_1,\emptyset) \Downarrow (3,\emptyset)$.
- Reste à comparer dynamique par valeur et statique par valeur. Si l'on se restreint à des expressions sans *quote*, *unquote* alors les deux stratégies coïncident. En effet dans la liaison statique on va associer à une variable un nombre et donc l'environnement ne joue pas de rôle. On modifie donc l'exemple ci-dessus comme suit :

  $$e_1 \equiv let\ x = 3\ in\ e_2, \quad e_2 \equiv let\ y = quote(x)\ in\ e_3, \quad e_3 \equiv let\ x = 5\ in\ unquote(y)\ .$$

  Maintenant $(e_1,\emptyset)$ s'évalue en 5 en dynamique par valeur et en $(3,\emptyset)$ en statique par valeur.

6.6 Typage

On dénote par *Types* l'ensemble des types et par *ProcTypes* les vecteurs de types qu'on associe aux procédures. Un *environnement de types* est une fonction

$$E : Id \cup Pid \rightharpoonup Types \cup ProcTypes$$

dont le domaine est fini avec $E(Id) \subseteq Types$ et $E(Pid) \subseteq ProcTypes$. Un type est une abstraction d'une valeur. De même, un environnement de types est une abstraction d'un environnement.

Typage des expressions On considère un jugement de la forme :

$$E \vdash e : \tau$$

et on suppose que dans une expression on nomme jamais explicitement une location ℓ. Les règles de typage sont les suivantes :

$$\frac{}{E \vdash true : bool} \qquad \frac{}{E \vdash false : bool} \qquad \frac{}{E \vdash n : int} \qquad \frac{E(x) = \tau}{E \vdash x : \tau}$$

$$\frac{E \vdash e_i : \tau_i \quad i = 1, 2}{E \vdash Pair(e_1, e_2) : \tau_1 * \tau_2} \qquad \frac{E \vdash e : \tau_1 * \tau_2}{E \vdash Fst(e) : \tau_1} \qquad \frac{E \vdash e : \tau_1 * \tau_2}{E \vdash Snd(e) : \tau_2}$$

$$\frac{E \vdash e : \tau}{E \vdash ref\ e : ref\ \tau} \qquad \frac{E \vdash e : ref\ \tau}{E \vdash !e : \tau}$$

Typage des environnements On considère des jugements de la forme :

$$E \vdash D : E' \qquad E \vdash DP : E'$$

Les règles de typage sont :

$$\frac{}{E \vdash \epsilon : E} \qquad \frac{E \vdash e : \tau \quad E[\tau/x] \vdash D : E'}{E \vdash (let\ x = e)\ ; D : E'}$$

$$\frac{E_{|Pid}[(\tau_1, \ldots, \tau_n)/f, \tau_1/x_1, \ldots, \tau_n/x_n] \vdash D; S \quad E[(\tau_1, \ldots, \tau_n)/f] \vdash DP : E''}{E \vdash (procedure\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) = D; S); DP : E''}$$

Typage des commandes On considère des jugements de la forme :

$$E \vdash S$$

Les règles de typage sont :

$$\frac{E(x) = ref\ \tau \quad E \vdash e : \tau}{E \vdash x := e} \qquad \frac{E \vdash e : bool \quad E \vdash S}{E \vdash while\ e\ do\ S} \qquad \frac{E \vdash e : bool \quad E \vdash S_1 \quad E \vdash S_2}{E \vdash if\ e\ then\ S_1\ else\ S_2}$$

$$\frac{E(f) = (\tau_1, \ldots, \tau_n) \quad E \vdash e_i : \tau_i \quad i = 1, \ldots, n}{E \vdash f(e_1, \ldots, e_n)} \qquad \frac{E \vdash S_1 \quad E \vdash S_2}{E \vdash S_1; S_2}$$

Typage des programmes On considère un jugement de la forme :

$$E \vdash P$$

La règle de typage est :

$$\frac{E \vdash DP : E' \quad E' \vdash D : E'' \quad E'' \vdash S}{E \vdash DP; D; S}$$

Exercice 6.14 *Typez, si possible, les programmes :*

$procedure\ f() = x :=!x + 1;$	$(procedure\ f\ (x : int, y : ref\, int) =$
$let\ x = ref\ 1;$	$let\ i = ref\ x;$
$f()$	$y := 1;\ (while\ !i > 0\ do\ (y :=!i * !y; i :=!i - 1)));$
	$let\ out = ref\ 0;\ let\ in = 2;$
	$f(in, out)$

Vérification de type L'application des règles de typage est dirigée par la syntaxe abstraite. Il s'agit simplement de fixer une représentation des environnements de type et d'écrire des fonctions récursives qui prennent en argument un environnement de type et vérifient le bon typage d'expressions, déclarations, commandes et programmes :

Propriétés du typage L'objectif des règles de typage est toujours de s'assurer que :

Un programme bien typé ne s'évalue pas en erreur.

La formulation de cette propriété et sa vérification est maintenant beaucoup plus compliquée que dans le langage d'expressions considéré dans la section 2. Par exemple, on peut commencer par traiter un fragment du langage où :

$$\begin{array}{lll} v & ::= & n \mid \ell \\ e & ::= & v \mid id \mid ref\ e \mid !e \\ D & ::= & \epsilon \mid (let\ id = e); D \\ P & ::= & D \end{array}$$

On remarquera que l'évaluation d'expressions peut produire des locations. On a donc besoin de 'typer les locations'. Plus en général, on a besoin d'exprimer les propriétés de la mémoire qui sont préservées par l'évaluation. On est donc amené à formuler une notion de *type de mémoire* qui abstrait une mémoire dans le même sens qu'un environnement de types abstrait un environnement.

Exercice 6.15 *On souhaite analyser les programmes suivants qui se composent d'une déclaration de procédure f, d'une déclaration de variable x et d'un corps principal $f()$.*

Programme 1	**Programme 2**	**Programme 3**
$procedure\ f() =$	$procedure\ f() =$	$procedure\ f() =$
$\quad let\ x = ref\ 1;$	$\quad let\ x = ref\ \mathsf{true};$	$\quad let\ y = ref\ !x;$
$\quad let\ y = ref\ !x;$	$\quad let\ y = ref\ 2\ ;;$	$\quad x :=!y;$
$\quad x :=!y;$	$\quad y :=!x;$	$let\ x = ref\ 1;$
$let\ x = ref\ \mathsf{true};$	$let\ x = ref\ 1;$	$f()$
$f()$	$f()$	

Pour chaque programme, précisez si :

1. *Le programme est bien typé.*
2. *L'exécution du programme est susceptible de produire une erreur.*

Suggestion *Vous pouvez répondre à ces questions sans calculer formellement le typage et l'évaluation des programmes.*

Exercice 6.16 *On étend la catégorie syntaxique S des commandes du langage avec une commande*

$$(\textit{repeat } S \textit{ until } e)\ .$$

La sémantique informelle de cette commande est la suivante : (A) On exécute la commande S. (B) On évalue l'expression e. (C) Si le résultat de l'évaluation est true alors on termine, si le résultat de l'évaluation est false alors on saute au point (A), autrement on avorte l'exécution et on donne un message d'erreur.

1. *Proposez un codage de la commande repeat dans le langage impératif avec while qui respecte la sémantique informelle (on peut supposer que le langage comprend un opérateur not qui calcule la négation sur le type des booléens).*
2. *Donnez les règles d'évaluation pour la commande repeat.*
3. *Donnez les règles de typage pour la commande repeat qui permettent d'éviter la situation d'erreur décrite dans la sémantique.*

Exercice 6.17 *On considère le langage impératif décrit dans le cours. On étend la catégorie syntaxique S des commandes avec une commande* for *de la forme*

$$(\textit{for } (id; S')\ S)\ .$$

dont la sémantique informelle est la suivante.

1. *On vérifie que la valeur associée à l'identificateur id est une référence ℓ et que la valeur v associée à la la référence ℓ est un entier.*
2. *Si v n'est pas un entier on donne un message d'erreur.*
3. *Si v est égal à 0 on termine l'exécution de la commande.*
4. *Autrement, on exécute la commande S suivie par la* commande d'incrément S' *et on saute au point 1.*

Vous devez :

1. *Donner les règles formelles d'évaluation pour la commande* for.
2. *Donner une règle de typage pour la commande* for *qui permet d'éviter la situation d'erreur décrite dans la sémantique informelle.*

7 Évaluation et typage d'un langage à objets

On suppose que le lecteur a déjà pratiqué la programmation à objets. On présente la syntaxe (plutôt abstraite), les règles d'évaluation et les règles de typage d'un micro-langage à objets librement inspiré du langage Java.

Un *objet* est constitué d'un nom de *classe* et d'une liste de locations de mémoire qu'on appelle *attributs*. Une *classe* est une déclaration dans laquelle on spécifie comment construire et manipuler les objets de la classe. En particulier, on spécifie les attributs de chaque objet et les méthodes qui permettent de les manipuler.

Les langages à objets reposent sur les notions d'*héritage* et de *sous-typage*. On rappelle que la première est une relation entre les *implémentations* des classes alors que la deuxième est une relation entre les *interfaces* des classes. Nous décrivons une approche élémentaire dans laquelle l'héritage est *simple* (au lieu d'être *multiple*) et la relation de sous-typage est dérivée de l'héritage (en général les deux notions peuvent être incomparables).

7.1 Syntaxe

Classes On suppose une classe *Object* sans attributs et sans méthodes. Chaque déclaration de classe étend une autre classe. Par exemple, on peut déclarer une classe C qui hérite de la classe D et qui comprend un attribut f et une méthode m.

$\mathsf{class}\ C\ \mathsf{extends}\ D\ = ($
$\cdots$
$\mathsf{var}\ f : C'$ (déclaration d'attribut *(field)*)
$\cdots$
$\mathsf{method} m(x_1 : D_1, ..., x_n : D_n) : D' = e$ (déclaration de méthode)
$\cdots)$

Les noms des classes seront les *types* de notre langage. On dérive de l'héritage une relation binaire de *sous-typage* $\leq$ sur les classes comme la plus petite relation réflexive et transitive telle que $C \leq D$ si le programme contient une déclaration de la forme ci-dessus. Pour interdire des héritages *cycliques*, on demande à ce que $C \leq D$ et $D \leq C$ implique $C = D$. Sous cette hypothèse, on peut représenter la relation de sous-typage comme un *arbre* ayant la classe *Object* à sa racine (la racine étant 'en haut').

Valeurs, locations et mémoires Nous allons supposer que tous les attributs sont *modifiables*. On reprend et on adapte les notions de *location* et de *mémoire* que nous avons considéré pour le langage impératif. Soit *Loc* un ensemble infini de locations avec éléments $\ell, \ell', \ldots$ Une location est maintenant un pointeur (ou référence) à un objet. La *valeur* v d'un objet prend la forme :

$$v ::= C(\ell_1, \ldots, \ell_n) \qquad n \geq 0$$

où C est le nom de la classe à laquelle l'objet appartient et $\ell_1, \ldots, \ell_n$ sont les locations associées aux n attributs modifiables de l'objet. Une mémoire μ est une fonction partielle à domaine fini qui associe à une location la valeur d'un objet.

Expressions et Commandes Les catégories syntaxiques des expressions e et des commandes S sont définies comme suit :

$$
\begin{array}{rcll}
e & ::= & x & | \text{ (variable)} \\
 & & v & | \text{ (valeur)} \\
 & & \mathsf{new}\ C(e_1, \ldots, e_n) & | \text{ (génération d'objet)} \\
 & & e.f & | \text{ (invocation d'attribut)} \\
 & & e.m(e_1, \ldots, e_n) & | \text{ (invocation de méthode)} \\
 & & (e\ \mathsf{as}\ C) & | \text{ (coercition/downcasting)} \\
 & & S; e & \text{(commande-expression)} \\
 & & & \\
S & ::= & e.f := e & | \text{ (affectation d'attribut)} \\
 & & S; S & \text{(séquentialisation)}
\end{array}
$$

Conventions Parmi les variables, on réserve la variable *this* (on choisit aussi *self*) pour faire référence à l'objet sur lequel la méthode est invoquée. Par ailleurs, comme dans le langage impératif, on peut faire l'hypothèse que le programme source ne manipule pas directement les locations.

Programme Un *programme* est constitué d'une liste de déclarations de classes, et d'une expression. La valeur de l'expression est le résultat du programme. Un programme bien formé doit satisfaire certaines conditions :

1. Si $C \leq D$ alors C hérite de tous les attributs de D. On demande à qu'il n'y ait pas de conflit de nom parmi les attributs. En d'autres termes, si on remonte un chemin de l'arbre d'héritage on ne doit pas trouver deux attributs avec le même nom.
2. Si $C \leq D$ alors C hérite aussi de toutes les méthodes de D, cependant C peut redéfinir (*override*) une méthode à condition que son type soit le même que le type de la méthode héritée.

Il convient d'introduire un certain nombre de fonctions qui seront utilisées dans la formulation des règles d'évaluation et de typage.

- *field*(C) retourne la liste $f_1 : C_1, \ldots, f_n : C_n$ des attributs accessibles par un objet de la classe C. Si on génère un objet de la classe C on doit donc lui passer n arguments.

- *mtype*(m, C) retourne le type de la méthode m dans la classe C.

- *override*$(m, D, \mathbf{C} \to C)$ est un prédicat qui vérifie que si *mtype*(m, D) est défini alors il coïncide avec $\mathbf{C} \to C$.

- *mbody*(m, C) retourne le corps $(x_1, \ldots, x_n, e)$ de la méthode m dans la classe C ; ici $x_1, \ldots, x_n$ sont les paramètres formels et e est l'expression associée à la méthode.

Exemple 7.1 *On considère une suite de déclarations de classes qui permettent de représenter les valeurs booléennes, le nombres naturels en notation unaire, les listes d'objets et les références.*

```
class Bool extends Object = (
      method ite(x:Object, y:Object): Object = new Object()  )
```

```
class True extends Bool = (
      method ite(x:Object,y:Object): Object = x )
class False extends Bool = (
      method ite(x:Object,y:Object): Object = y )
class Num extends Object = (
      method iszero():Bool = new Bool()
      method pred():Num = new Num() )
class Zero extends Num = (
      method iszero():Bool = new True() )
class NotZero extends Num = (
      var pd: Num
      method pred() : Num = this.pd
      method iszero(): Bool = new False() )
class List extends Object = (
      method select(n:Num): Object = new Object()
      method insert(c:Object): NotEmptyList = new NotEmptyList(c,this) )
class NotEmptyList extends List = (
      var cl: Object
      var follow: List
      method select(n:Num): Object =  n.iszero().ite( this.cl, this.follow.select(n.pred())) )
class Ref extends Object = (
      var val : Object
      read(): Object = this.val
      write(x:Object):Object = this.val:=x; this )
```

Exercice 7.2 *Complétez le code de l'exemple 7.1 en écrivant les méthodes suivantes :*

1. *Une méthode* cpl *dans la classe* Bool *pour calculer le complémentaire d'un booléen.*
2. *Une méthode* add *dans la classe* Num *avec un argument de type* Num *pour additionner.*
3. *Une méthode* count *dans la classe* List *pour compter le nombre d'éléments d'une liste.*

7.2 Évaluation

Pour se rapprocher d'une mise-en-oeuvre de l'évaluateur, il conviendra de formuler les règles d'évaluation par rapport à un *environnement* η qui associe des valeurs aux variables. Le jugements qu'on considère ont la forme : $(e, \eta, \mu) \Downarrow (v, \mu')$ et $(S, \eta, \mu) \Downarrow \mu'$. On évalue les expressions et les commandes par rapport à une mémoire et un environnement. Dans le premier cas le résultat de l'évaluation est une valeur et une nouvelle mémoire et dans le deuxième le résultat est une mémoire. Les règles d'évaluation sont les suivantes, où l'on suppose que $new(\mu, n) = \ell_1, \ldots, \ell_n$ si $\ell_1, \ldots, \ell_n$ sont n locations 'nouvelles' (pas déjà utilisées dans μ).

$$\frac{}{(v,\eta,\mu) \Downarrow (v,\mu)} \qquad \frac{}{(x,\eta,\mu) \Downarrow (\eta(x),\mu)}$$

$$\frac{(e_1,\eta,\mu) \Downarrow (v_1,\mu_1), \ldots, (e_n,\eta,\mu_{n-1}) \Downarrow (v_n,\mu_n), \quad \ell_1,\ldots,\ell_n = new(\mu,n)}{(\mathsf{new}\ C(e_1,\ldots,e_n),\eta,\mu) \Downarrow (C(\ell_1,\ldots,\ell_n), \mu_n[v_1,\ldots,v_n/\ell_1,\ldots,\ell_n])}$$

$$\frac{(e,\eta,\mu) \Downarrow (C(\ell_1,\ldots,\ell_n),\mu') \quad \mathit{field}(C) = f_1 : C_1,\ldots,f_n : C_n \quad 1 \le i \le n}{(e.f_i,\eta,\mu) \Downarrow (\mu'(\ell_i),\mu')}$$

$$\frac{(e,\eta,\mu) \Downarrow (C(\ell),\mu_0) \quad \mathit{mbody}(m,C) = (x_1,\ldots,x_n,e') \quad (e_1,\eta,\mu_0) \Downarrow (v_1,\mu_1)\cdots(e_n,\eta,\mu_{n-1}) \Downarrow (v_n,\mu_n) \quad (e',\eta[v_1,\ldots,v_n/x_1,\ldots,x_n,C(\ell)/\mathit{this}],\mu_n) \Downarrow (v,\mu')}{(e.m(e_1,\ldots,e_n),\eta,\mu) \Downarrow (v,\mu')}$$

$$\frac{(e,\eta,\mu) \Downarrow (C(\ell),\mu') \quad C \le D}{(e\ \mathsf{as}\ D,\eta,\mu) \Downarrow (C(\ell),\mu')} \qquad \frac{(S,\eta,\mu) \Downarrow \mu' \quad (e,\eta,\mu') \Downarrow (v,\mu'')}{(S;e,\eta,\mu) \Downarrow (v,\mu'')}$$

$$\frac{(e,\eta,\mu) \Downarrow (C(\ell_1,\ldots,\ell_n),\mu') \quad \mathit{field}(C) = f_1 : C_1,\ldots,f_n : C_n \quad (e',\eta,\mu') \Downarrow (v,\mu'')}{(e.f_i := e',\eta,\mu) \Downarrow \mu''[v/\ell_i]} \qquad \frac{(S,\eta,\mu) \Downarrow \mu' \quad (S',\eta,\mu') \Downarrow \mu''}{(S;S',\eta,\mu) \Downarrow \mu''}$$

Exemple 7.3 *On se réfère aux classes définies dans l'exemple 7.1. Soient η un environnement vide, μ une mémoire vide et e l'expression* new Ref(new True()).write(new False())).read(). *On peut vérifier que l'évaluation de (e,η,μ) produit l'expression* False() *et la mémoire μ*[False()/ℓ].

7.3 Typage

Un objectif général des systèmes de typage pour les langages à objets est de garantir que chaque invocation d'un attribut ou d'une méthode sur un objet est bien compatible avec la classe à laquelle l'objet appartient. Notons cependant qu'une mauvaise utilisation de la coercition (*downcasting*) peut compromettre cette propriété. Par exemple, on pourrait écrire l'expression :

$$(\mathsf{new}\ \mathsf{Object}()\ \mathsf{as}\ \mathsf{Ref}).\mathsf{read}()$$

Pour éviter cette situation on pourrait envisager une règle de la forme :

$$\frac{E \vdash e : D \quad D \le C}{E \vdash (e\ \mathsf{as}\ C) : C}$$

Cependant cette règle s'avère trop contraignante. Par exemple, elle nous empêche de typer l'expression :

$$(\mathsf{new}\ \mathsf{True}()).\mathsf{ite}(\mathsf{new}\ \mathsf{True}(),\ \mathsf{new}\ \mathsf{False}\ ())\ \mathsf{as}\ \mathsf{Bool}$$

car le résultat de la méthode ite appartient à la classe Object et Object $\not\le$ Bool.

En Java la règle pour la coercition est plutôt :

$$\frac{E \vdash e : D \qquad (C \le D \text{ ou } D \le C)}{E \vdash (e\ \mathsf{as}\ C) : C}$$

En d'autres termes, la coercition descendante est *interdite* si C et D sont *incomparables*. Cependant, cette propriété n'est pas préservée par évaluation ! Soient A, B deux classes incomparables et e une expression de type A. Alors l'expression $((e \text{ as } Object) \text{ as } B)$ est bien typée mais elle se simplifie en l'expression $(e \text{ as } B)$ qui ne l'est plus. En montant et descendant dans l'arbre d'héritage, on peut arriver à des classes incomparables...

Pour cette raison, on écrit la règle de typage pour la coercition descendante comme suit :

$$\frac{E \vdash e : D}{E \vdash (e \text{ as } C) : C}$$

Au moment du typage, on ne cherche pas a vérifier que la valeur $C'(\ell)$ qui résulte de l'évaluation de l'expression e est telle que $C' \leq C$. En effet, on retarde cette vérification jusqu'au moment de l'évaluation de la coercition. Si la condition n'est pas satisfaite on bloque l'évaluation (alternativement, on pourrait produire un message d'erreur).

Typage d'expressions et de commandes Un environnement de types E a la forme $x_1 : C_1, \ldots, x_n : C_n$. On considère les jugements de typage de la forme :

$$E \vdash e : C, \qquad E \vdash S \ .$$

On suppose que dans une expression on ne nomme jamais explicitement une location.

$$\frac{x : C \in E}{E \vdash x : C} \qquad \frac{\mathit{field}(C) = f_1 : D_1, \ldots, f_n : D_n \quad E \vdash e_i : C_i, \quad C_i \leq D_i, \quad 1 \leq i \leq n}{E \vdash \text{new } C(e_1, \ldots, e_n) : C}$$

$$\frac{E \vdash e : C \quad \mathit{field}(C) = f_1 : C_1, \ldots, f_n : C_n}{E \vdash e.f_i : C_i} \qquad \frac{E \vdash e : C \quad \mathit{mtype}(m, C) = (C_1, \ldots, C_n) \to D \quad E \vdash e_i : C_i' \quad C_i' \leq C_i \quad 1 \leq i \leq n}{E \vdash e.m(e_1, \ldots, e_n) : D}$$

$$\frac{E \vdash e : D}{E \vdash (e \text{ as } C) : C} \qquad \frac{E \vdash S \qquad E \vdash e : C}{E \vdash S; e : C}$$

$$\frac{E \vdash e : C \quad \mathit{field}(C) = f_1 : C_1, \ldots, f_n : C_n \quad E \vdash e' : D_i \quad D_i \leq C_i}{E \vdash e.f_i := e'} \qquad \frac{E \vdash S_1 \qquad E \vdash S_2}{E \vdash S_1; S_2}$$

Remarque 7.4 (sous-typage) *On remarquera que les règles de typage permettent d'utiliser un objet de la classe C là où on attend un objet de la classe D à condition que C soit une sous-classe de D.*

Typage de méthodes, de classes et de programmes Une méthode m de la forme

$$m(x_1 : C_1, \ldots, x_n : C_n) : C_0 = e$$

dans une classe C qui hérite de la classe D est bien typée si :

1. $\mathit{override}(m, D, (C_1, \ldots, C_n) \to C_0)$,
2. $x_1 : C_1, \ldots, x_n : C_n, \mathit{this} : C \vdash e : C_0'$ et $C_0' \leq C_0$.

Une classe est bien typée si toutes ses méthodes sont bien typées. Un programme est bien typé si toutes ses classes sont bien typées et si l'expression résultat est bien typée dans l'environnement de type vide.

Exemple 7.5 *On peut typer les classes définies dans l'exemple 7.1 et l'expression définie dans l'exemple 7.3. On remarque qu'on utilise les sous-typage* $\mathsf{True} \leq \mathsf{Object}$ *au moment de la création de l'objet* Ref *et le sous-typage* $\mathsf{False} \leq \mathsf{Object}$ *au moment de l'invocation de la méthode* write. *Par ailleurs, le type de l'expression e définie est* Object.

Exercice 7.6 *Définissez une classe Bnum des nombres naturels en notation* binaire *avec des méthodes pour incrémenter, décrémenter, additionner et tester-le-zéro.*

Exercice 7.7 *On considère un fragment du langage à objets auquel on a ajouté les expressions* fail *et* $\mathsf{catch}(e, e')$ *:*

$$e ::= id \mid \mathsf{new}\ C(e, \ldots, e) \mid (e\ \mathsf{as}\ C) \mid \mathsf{fail} \mid \mathsf{catch}(e, e')$$

Un jugement d'évaluation pour les expressions a la forme $(e, \eta, \mu) \Downarrow (u, \mu')$ *où* η *est un environnement,* μ *et* μ' *sont des mémoires et u est ou bien une valeur ou bien* fail *(donc* fail *n'est pas une valeur).*
Proposez des règles d'évaluation pour les expressions qui respectent les conditions suivantes :

- *L'évaluation des expressions sans* fail *et* catch *se passe normalement.*
- *L'expression* $(D()\ \mathsf{as}\ C)$ *s'évalue en* fail *si* D *n'est pas une sous-classe de* C.
- *Si l'expression e s'évalue en une valeur alors l'expression* $\mathsf{catch}(e, e')$ *s'évalue comme e.*
- *L'expression* $\mathsf{catch}(\mathsf{fail}, e')$ *s'évalue comme* e' *(L'idée est qu'un échec provoqué par* fail *se propage et entraîne l'arrêt du programme sauf s'il est traité par un* catch*).*

On dispose de deux classes C *et* D *sans attributs et telles que* $C \leq D$ *mais* $D \not\leq C$.
Utilisez vos règles pour évaluer les expressions suivantes dans un environnement et une mémoire vides :

$$\begin{array}{lcl} e_1 & = & \mathsf{catch}(\ \ (\mathsf{new}\ D())\ \mathsf{as}\ C,\ \ \mathsf{new}\ C()\ \ ) \\ e_2 & = & \mathsf{catch}(\ \ \mathsf{fail},\ \ (\mathsf{new}\ C())\ \mathsf{as}\ D\ \ ) \\ e_3 & = & \mathsf{catch}(\ \ \mathsf{catch}(\ \mathsf{fail},\ \mathsf{new}\ C()\ ),\ \ \mathsf{new}\ D()\ \ ) \end{array}$$

8 Évaluation et typage d'un langage fonctionnel

On suppose que le lecteur est familier avec un langage fonctionnel de la famille `ML`. On considère un langage d'expressions dans lequel on peut manipuler des fonctions de fonctions. On dit que le langage est d'*ordre supérieur*. Il s'agit d'un langage minimal introduit par Church en 1930 qu'on appelle 'λ-calcul'.

$$\begin{array}{ll} id & ::= x \mid y \mid \ldots \\ e & ::= id \mid (\lambda id.e) \mid (ee) \end{array}$$

Les seules opérations du langage sont l'*abstraction* $\lambda x.e$ et l'*application* ee'. [4] Une variété d'autres opérations peuvent être vues comme du *sucre syntaxique*. Par exemple, l'opération $\mathsf{let}\ x = e\ \ \mathsf{in}\ e'$ est représentée par $(\lambda x.e')e$.

8.1 Substitution

L'abstraction $\lambda x.e$ lie la variable x dans le terme e exactement comme dans la formule du premier ordre $\forall x.\phi$ le quantificateur universel lie x dans ϕ. On dénote par $FV(e)$ l'ensemble des variables qui paraissent libres dans le terme e. On dit que deux termes e, e' sont α-équivalents, et on écrit $e =_\alpha e'$ si on peut obtenir l'un de l'autre par renommage des variables liées. Par exemple, $\lambda f.\lambda x.f(fx) =_\alpha \lambda x.\lambda y.x(xy)$.

A cause de la présence de variables liées, la substitution $[e'/x]e$ doit être définie avec un peu d'attention (cf. section 2). Comment définir $[e'/x](\lambda y.e)$ si $x \neq y$ et $y \in FV(ee')$? Une définition possible est la suivante :

$$\begin{array}{ll} [e'/x]x & = e' \\ [e'/x]y & = y \text{ si } y \neq x \\ [e'/x](e_1 e_2) & = [e'/x]e_1[e'/x]e_2 \\ [e'/x](\lambda y.e) & = \lambda z.[e'/x][z/y]e \text{ si } z \notin FV(ee') \end{array}$$

Pour vérifier qu'il s'agit bien d'une définition inductive sur la structure d'un λ-terme, il faut d'abord noter que $[z/y]e$ a la même taille que e. Le lecteur peut aussi vérifier que (i) $[e'/x](\lambda x.e) =_\alpha \lambda x.e$ et (ii) si $x \neq y$ et $y \notin FV(e')$ alors $[e'/x](\lambda y.e) =_\alpha \lambda y.[e'/x]e$.

8.2 Appel par nom et appel par valeur

Les valeurs $v, v', \ldots$ sont les λ-termes définis par la grammaire :

$$v ::= \lambda id.e\ .$$

On spécifie dans la table 2, les relations $\Downarrow_N$ et $\Downarrow_V$ qui définissent l'évaluation pour l'appel par nom et par valeur, respectivement (la liaison étant toujours statique, cf section 6).

8.3 Typage

On définit la collection des types par la grammaire :

$$\tau ::= b \mid Tid \mid (\tau \to \tau)$$

[4] En ML, on écrit (function $x \to e$) pour $\lambda x.e$

$$\frac{}{v \Downarrow_N v} \qquad \frac{e \Downarrow_N \lambda x.e_1 \quad [e'/x]e_1 \Downarrow_N v}{ee' \Downarrow_N v}$$

$$\frac{}{v \Downarrow_V v} \qquad \frac{e \Downarrow_V \lambda x.e_1 \quad e' \Downarrow_V v' \quad [v'/x]e_1 \Downarrow_V v}{ee' \Downarrow_V v}$$

TAB. 2 – Évaluation en appel par nom et par valeur

où $Tid ::= t \mid s \mid \ldots$ Un environnement de type E est toujours une fonction à domaine fini des variables aux types qu'on représente aussi comme une liste de couples $x_1 : \tau_1, \ldots, x_n : \tau_n$ où toutes les variables $x_1, \ldots, x_n$ sont différentes. On écrit $E, x : \tau$ pour la fonction $E[\tau/x]$ où x n'est pas dans le domaine de définition de E. Les règles de typage sont les suivantes :

$$(ax) \quad \frac{E(x) = \tau}{E \vdash x : \tau}$$

$$(\rightarrow_I) \quad \frac{E[\tau/x] \vdash e : \tau'}{E \vdash \lambda x.e : \tau \rightarrow \tau'} \qquad (\rightarrow_E) \quad \frac{E \vdash e : \tau \rightarrow \tau' \quad E \vdash e' : \tau}{E \vdash ee' : \tau'}$$

Nous vérifions que le typage est préservé par la relation d'évaluation (cf. section 2). D'abord on a besoin d'un lemme.

Lemme 8.1 (substitution) *Si $E, x : \tau \vdash e : \tau'$ et $E \vdash e' : \tau$ alors $E \vdash [e'/x]e : \tau'$.*

IDÉE DE LA PREUVE. Par induction sur la hauteur de la preuve de $E, x : \tau \vdash e : \tau'$. Par exemple, supposons que la racine de l'arbre de preuve ait la forme :

$$\frac{E, x : \tau, y : \tau' \vdash e : \tau''}{E, x : \tau \vdash \lambda y.e : \tau' \rightarrow \tau''}$$

avec $x \neq y$. Par hypothèse de récurrence, $E, y : \tau' \vdash [e'/x]e : \tau''$ et on conclut par $(\rightarrow_I)$. •

Proposition 8.2 (réduction du sujet) *Si $E \vdash e : \tau$ et $e \Downarrow_S v$ où $S \in \{N, V\}$ alors $E \vdash v : \tau$.*

IDÉE DE LA PREUVE. Par induction sur la hauteur de la preuve de $e \Downarrow_S v$. On considère le cas où l'évaluation est par valeur et la racine de la preuve a la forme :

$$\frac{e \Downarrow_V \lambda x.e_1 \quad e' \Downarrow_V v' \quad [v'/x]e_1 \Downarrow_V v}{ee' \Downarrow_V v}$$

Alors $E \vdash ee' : \tau$ implique $E \vdash e : \tau' \rightarrow \tau$ et $E \vdash e' : \tau'$ pour quelque τ'. Par hypothèse inductive, $E \vdash \lambda x.e_1 : \tau' \rightarrow \tau$ et $E \vdash v' : \tau'$. Alors on dérive que $E, x : \tau' \vdash e_1 : \tau$. Par le lemme de substitution $E \vdash [v'/x]e_1 : \tau$, et par hypothèse inductive $E \vdash v : \tau$. •

Étant donné un terme e et un contexte E, le *problème d'inférence de type* est le problème de vérifier qu'il y a un type τ tel que $E \vdash e : \tau$. Étant donné un terme e, une variante du problème est de chercher un type τ *et* un contexte E tel que $E \vdash e : \tau$.

Associé au problème de l'inférence de type, on a le problème de produire une information intéressante. Si un terme e est typable, on cherche une représentation synthétique de ses types et s'il ne l'est pas on souhaite émettre un message d'erreur informatif.

Exercice 8.3 *Montrez que si* $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$ *est dérivable alors* $(\tau_1 \to \cdots (\tau_n \to \tau) \cdots)$ *est une tautologie de la logique propositionnelle où l'on interprète* $\to$ *comme une implication et les types atomiques comme des variables propositionnelles.*

Exercice 8.4 *Trouvez un type* τ *tel que* $\emptyset \vdash \lambda x.\lambda y.x(yx) : \tau$ *est dérivable et explicitez la dérivation.*

Exercice 8.5 *On considère le type :*

$$\sigma \equiv (((\tau \to \tau') \to \tau') \to \tau') \to (\tau \to \tau')$$

Présentez un λ*-terme fermé (c.a.d. sans variables libres)* e *tel que* $\emptyset \vdash e : \sigma$ *est dérivable dans le système ci-dessus.*

Même question pour le type :

$$(\tau_1 \to \tau_2) \to ((\tau_2 \to \tau_3) \to (\tau_1 \to \tau_3))$$

où τ_1, τ_2, τ_3 *sont trois types* différents.

8.4 Un évaluateur pour le langage fonctionnel

Nous allons considérer plus en détail un évaluateur pour le λ-calcul. On omet les types et on se focalise sur les stratégies d'évaluation en appel par nom et par valeur. Nous allons raffiner la description de l'évaluateur en utilisant le concept de *clôture* que nous avons déjà évoqué dans la section 6.5. On définit environnements et clôtures de la façon suivante :

- Un *environnement* est une fonction partielle $\eta : \mathit{Var} \rightharpoonup \mathit{Closures}$ où $\mathit{dom}(\eta)$ est fini (en particulier la fonction qui est toujours indéfinie est un environnement), et *Closures* est l'ensemble des clôtures.
- Une *clôture* c est un couple qu'on dénote par $e[\eta]$ où e est un terme et η est un environnement.

On évalue toujours des clôtures $e[\eta]$ telles que $FV(e) \subseteq \mathit{dom}(\eta)$.

On peut reformuler les relations d'évaluation sur les clôtures. Maintenant, une valeur vc est une clôture de la forme $(\lambda x.e)[\eta]$. Les règles sont présentées dans la table 3.

$$\frac{}{vc \Downarrow_N vc} \qquad \frac{\eta(x) \Downarrow_N vc}{x[\eta] \Downarrow_N vc} \qquad \frac{e[\eta] \Downarrow_N \lambda x.e_1[\eta'] \quad e_1[\eta'[e'[\eta]/x]] \Downarrow_N vc}{(ee')[\eta] \Downarrow_N vc}$$

$$\frac{}{vc \Downarrow_V vc} \qquad \frac{\eta(x) \Downarrow_V vc}{x[\eta] \Downarrow_V vc} \qquad \frac{e[\eta] \Downarrow_V \lambda x.e_1[\eta'] \quad e'[\eta] \Downarrow_V vc' \quad e_1[\eta'[vc'/x]] \Downarrow_V vc}{(ee')[\eta] \Downarrow_V vc}$$

TAB. 3 – Évaluation des clôtures en appel par nom et par valeur

Exercice 8.6 *On voit un environnement η comme une liste de couples $(x_1, c_1), \cdots, (x_n, c_n)$ où x_i sont des variables et c_i des clôtures. Nous allons ré-écrire les clôtures en éliminant les variables qui suivent immédiatement un 'λ' (les paramètres formels) et en remplaçant les autres variables (les occurrences dans les corps des fonctions) par des* indices *(des nombres naturels) qui indiquent la* distance *entre la variable et la λ-abstraction correspondante ou l'élément de l'environnement correspondant. Par exemple, en dénotant avec $\_$ la liste vide, on a la correspondance suivante :*

Syntaxe clôture avec variables	*Syntaxe clôture avec indices*
$((\lambda x.x)(\lambda x.x))[\_]$	$((\lambda.0)(\lambda.0))[\_]$
$((\lambda x.x)(\lambda y.y))[\_]$	$((\lambda.0)(\lambda.0))[\_]$
$(\lambda x.xy)[(z, (\lambda w.w)[\_]), (y, (\lambda w.w)[\_])]$	$(\lambda.02)[(\lambda.0)[\_], (\lambda.0)[\_]]$
$(\lambda x.xy)[(y, (\lambda w.w)[\_]), (z, (\lambda w.w)[\_])]$	$(\lambda.01)[(\lambda.0)[\_], (\lambda.0)[\_]]$

En utilisant cette nouvelle syntaxe, on peut ré-écrire les règles d'évaluation en appel par valeur de la façon suivante où $\eta(i)$ dénote l'i-ème élément de la liste η (on compte à partir de 0) :

$$\frac{}{(\lambda.e)[\eta] \Downarrow (\lambda.e)[\eta]} \qquad \frac{}{i[\eta] \Downarrow \eta(i)} \qquad \frac{e[\eta] \Downarrow \lambda.e_1[\eta_1] \quad e'[\eta] \Downarrow v_2[\eta_2] \quad e_1[v_2[\eta_2]\ ,\ \eta_1] \Downarrow v_3[\eta_3]}{(ee')[\eta] \Downarrow v_3[\eta_3]}$$

1. *Évaluez* $((\lambda.0)(\lambda.0))[\_]$.
2. *Programmez l'évaluateur dans le langage à objets présenté en section 7. Plus précisément vous devez programmer dans ce langage :*
   - *Une classe* Closure *avec attributs de type* Code *et* List *et une méthode* eval *qui retourne comme résultat l'évaluation de la clôture.*
   - *Une classe* Code *avec une méthode* ev *qui prend en argument un environnement (une liste) et retourne le résultat de l'évaluation du code par rapport à l'environnement. Par ailleurs la classe* Code *a comme sous-classes les classes* Var, Lambda *et* Apl.

   Il est conseillé d'utiliser les déclarations de classe Bool, Num *et* List *de l'exemple 7.1. Dans votre programme, vous pouvez utiliser la notation* let x=e in e' *comme abréviation pour l'expression* e' *où chaque occurrence de* x *est remplacée par l'expression* e. *Notez que la méthode* ite *de la classe* Bool *évalue toujours les deux branches de l'if-then-else.*
3. *Précisez si l'opération de* downcasting *joue un rôle dans le bon typage du programme.*
4. *Construisez l'expression* e *de type* Closure *qui correspond à la clôture* $((\lambda.0)(\lambda.0))[\_]$ *et vérifiez qu'elle est bien typée.*

8.5 Vers une mise en oeuvre

Nous allons raffiner encore les évaluateurs décrits dans la table 3 en introduisant une *pile* qui maintient une trace des valeurs calculées et des termes à évaluer.

Dans la stratégie en appel par nom, on visite un terme en cherchant une réduction possible à gauche de l'application. Pendant cette visite, les termes qui paraissent comme arguments d'une application sont insérés avec leurs environnements sur la pile.

Donc la pile (ou *stack*) s peut être vue comme une liste éventuellement vide de clôtures $c_1 : \ldots : c_n$.

On décrit le calcul comme un système de réécriture de couples $(e[\eta], s)$ composées d'une clôture et d'une pile. La machine opère sur des termes clos. Au début du calcul la pile est

vide et l'environnement est la fonction indéfinie partout.

$$\begin{array}{lcl}(x[\eta], s) & \rightarrow & (\eta(x), s)\\ ((ee')[\eta], s) & \rightarrow & (e[\eta], e'[\eta] : s)\\ ((\lambda x.e)[\eta], c : s) & \rightarrow & (e'[\eta[c/x]], s)\end{array}$$

Dans l'appel par valeur, on a besoin de savoir si le sommet de la pile est une fonction ou un argument. Pour cette raison, on insère dans la pile des marqueurs l (pour *left*) et r (pour *right*) qui spécifient si la prochaine clôture sur la pile est un argument gauche ou droite de l'application. La pile devient alors une liste éventuellement vide de marqueurs $m \in \{l, r\}$ et clôtures de la forme : $m_1 : c_1 : \ldots m_n : c_n$. Le calcul est maintenant décrit par les règles suivantes :

$$\begin{array}{lcl}(x[\eta], s) & \rightarrow & (\eta(x), s)\\ ((ee')[\eta], s) & \rightarrow & (e[\eta], r : e'[\eta] : s)\\ (vc, r : c : s) & \rightarrow & (c, l : vc : s)\\ (vc, l : (\lambda x.e)[\eta] : s) & \rightarrow & (e[\eta[vc/x]], s)\end{array}$$

8.6 Mise en oeuvre de l'évaluateur

Nous décrivons une mise en oeuvre de l'évaluateur pour l'appel par valeur. L'évaluateur gère une mémoire qui est divisée en trois parties (voir aussi section 9) :

Statique Cette partie contient :
- Les instructions à exécuter.
- Un pointeur pt_code à la prochaine instruction à exécuter. Initialement ce pointeur pointe à la première instruction.
- Un pointeur pt_stack au sommet de la pile (voir ci-dessous). Initialement ce pointeur pointe à la base de la pile.
- Un pointeur pt_env à l'environnement courant (qui est mémorisé dans le tas, voir ci-dessous). Initialement ce pointeur est nil.
- Un pointeur pt_free à la première cellule libre du tas.

Pile Une zone contiguë de mémoire dont le sommet est pointé par pt_stack. Initialement la pile est vide.

Tas Une zone contiguë de mémoire. Initialement cette zone est liée pour former une liste de *cellules libres* et le premier élément de la liste est pointé par pt_free.

Les instructions de la partie statique, les éléments de la pile et les éléments du tas sont structurés comme des enregistrements (ou *records*) avec les champs suivants :

enregistrement code Il a trois champs : op l'étiquette de l'instruction, left le pointeur gauche et right le pointeur droit.

enregistrement pile Il a trois champs : marqueur m, pointeur au code code, pointeur à l'environnement env.

enregistrement tas Il a quatre champs : var nom de la variable, code pointeur au code, env pointeur à l'environnement et next pointeur au prochain élément du tas.

La description de l'évaluateur peut être complétée pour traiter les problèmes du débordement de la pile, du débordement du tas et de la récupération de la mémoire du tas. On remarque que la fonction Eval utilise seulement des goto's ; en particulier, il n'y a pas d'appel récursif et donc il n'y a pas de pile *cachée* qui gère la récursion.

```
Eval :   case pt_code.op of

@ :      let p = push() in
         p.code := pt_code.right;
         p.env := pt_env;
         p.m := r;
         pt_code := pt_code.left;
         goto Eval

x :      let p = access(x, pt_env) in
         pt_code := p.code;
         pt_env := p.env;
         goto Eval

λx :     case
         pt_stack = ∅ : return(pt_code, pt_env)

         pt_stack.m = r :
         aux1 := pt_stack.code;
         aux2 := pt_stack.env;
         pt_stack.code := pt_code;
         pt_stack.env := pt_env;
         pt_stackpile.m := l;
         pt_code := aux1;
         pt_env := aux2;
         goto Eval

         pt_stack.m = l, pt_stack.code.op = λy :
         let p = pop(pt_free) in
         p.code := pt_code;
         p.env := pt_env;
         p.var := y;
         p.next := pt_stack.env;
         pt_code := pt_stack.code.right;
         pt_env := p;
         pop(pt_stack);
         goto Eval
```

TAB. 4 – Mise en oeuvre de l'évaluateur pour l'appel par valeur

Remarque 8.7 *L'évaluateur que nous venons de décrire n'est pas très éloigné d'une machine virtuelle pour un langage fonctionnel. En particulier, la machine virtuelle d'un langage fonctionnel manipule aussi une pile et un tas et la gestion des liaisons est basée sur la notion de clôture.*

Exercice 8.8 *Les règles suivantes décrivent une variante —avec* appel par nom *— de l'évaluateur pour l'appel par valeur :*

$$\begin{array}{lllll} (1) & (x[\eta], s) & \rightarrow & (\eta(x), s) \\ (2) & ((ee')[\eta], s) & \rightarrow & (e[\eta], e'[\eta] : s) \\ (3) & ((\lambda x.e)[\eta], c : s) & \rightarrow & (e[\eta[c/x]], s) \end{array}$$

où (i) $x, ee', \lambda x.e$ sont des λ-termes, (ii) η est un environnement c.a.d. une fonction partielle qui associe une clôture à une variable, (iii) c est une clôture c.a.d. un couple λ-terme-environnement (écrit $e[\eta]$) et (iv) s est une pile de clôtures.

1. *Évaluez la configuration $(e[\emptyset], \emptyset)$ où $e \equiv (((\lambda x.\lambda y.y)\Omega)I)$, $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ et $I \equiv \lambda z.z$.*
2. *Le résultat change-t-il si l'on utilise les règles pour l'appel par valeur ?*
3. *En vous inspirant de l'implémentation pour l'appel par valeur, décrivez la partie de l'implementation de l'évaluateur qui correspond à la règle (2). On suppose que* pt_code *est le pointeur au code,* pt_env *est le pointeur à l'environnement et* pt_stack *est le pointeur à la pile.*

9 Machine virtuelle et compilation

On décrit une machine virtuelle et une fonction de compilation pour le langage impératif. La mémoire de la machine virtuelle est organisée en 3 parties :

1. Une partie qui contient le code du programme. Cette partie est allouée statiquement et elle n'est pas modifiée. Le code est naturellement divisé en segments où chaque segment correspond à une procédure ou au corps principal du programme.
2. Une pile de blocs d'activation (ou *frames*). Un bloc d'activation est un triplet
$$(f, pc, v_1 \cdots v_n)$$
où :
   - f est l'adresse d'un segment de code qui correspond à la procédure f,
   - pc est un compteur ordinal qui varie sur les instructions du segment (à partir de 1) et
   - $v_1 \cdots v_n$ est une pile de valeurs (le sommet est à droite).

   Les blocs d'activation sont empilés selon l'ordre d'appel des procédures. Ainsi le bloc au sommet correspond au dernier appel qui est actuellement en exécution. Au début du calcul, le bloc d'activation sur la pile est celui du corps principal du programme.
3. Un tas (ou *heap*) qui est une association entre locations et valeurs.

Remarque 9.1 *On dispose d'une pile de blocs d'activation et dans chaque bloc d'activation on dispose d'une pile de valeurs. Attention à ne pas confondre les piles !*

9.1 Instructions du code octet

La machine virtuelle exécute un cycle standard de chargement exécution (*fetch and execute*). Si $(f, pc, v_1 \cdots v_n)$ est le bloc d'activation au sommet de la pile, la machine virtuelle exécute l'instruction d'adresse pc dans le segment d'instructions qui correspond à f. Les instructions et leur effet sur les piles sont décrites ci-dessous.

- build c n : on remplace n valeurs $v_1 \cdots v_n$ au sommet de la pile par $c(v_1, \ldots, v_n)$ et on incrémente le compteur ordinal. Ici c est un constructeur. Par exemple, on peut écrire : (build *true* 0) ou (build *Pair* 2).
- load n : on copie l'n-ième valeur de la pile au sommet de la pile (de valeurs) et on incrémente le compteur ordinal.
- goto j : on affecte j au compteur ordinal.
- branch j : si la valeur au sommet de la pile est *true* on incrémente le compteur ordinal sinon on affecte le compteur ordinal à j. Dans les deux cas on supprime la valeur au sommet de la pile.
- op n : on remplace les n valeurs $v_1 \cdots v_n$ au sommet de la pile par $\underline{op}(v_1, \ldots, v_n)$ et on incrémente le compteur ordinal.
- fst (snd) : si la valeur au sommet de la pile est $\mathsf{Pair}(v_1, v_2)$ alors on remplace cette valeur par v_1 (v_2) et on incrémente le compteur ordinal.
- stop : on arrête le calcul.
- call f n : on enlève les n valeurs $v_1 \cdots v_n$ au sommet de la pile, on incrémente le compteur ordinal et on empile un bloc d'activation $(f, 1, v_1 \cdots v_n)$. Cette instruction est utilisée dans l'appel de procédure.

– tcall f n : on sélectionne les n valeurs $v_1 \cdots v_n$ au sommet de la pile et on remplace le bloc d'activation courant par le bloc $(f, 1, v_1 \cdots v_n)$. Cette instruction peut être utilisée dans l'appel d'une procédure dont la récursion est terminale.
– return on dépile un bloc d'activation.
– new on génère une nouvelle location ℓ, on copie la valeur au sommet de la pile dans la location, on remplace la valeur par ℓ et on incrémente le compteur ordinal.
– read si la valeur au sommet de la pile est une location ℓ, on remplace ℓ par son contenu et on incrémente le compteur ordinal.
– write si les valeurs au sommet de la pile sont $v \cdot \ell$ alors on écrit v dans la location ℓ, on élimine v et ℓ et on incrémente le compteur ordinal.

On peut formaliser la sémantique des instructions par des règles de réécriture. Par exemple :

– La règle pour l'instruction new est :

$$(S \cdot (f, pc, \mathbf{v} \cdot v), \mu) \rightarrow (S \cdot (f, pc + 1, \mathbf{v} \cdot \ell), \mu[v/\ell])$$

où $f[pc] =$ new, S est une pile (éventuellement vide) de blocs d'activation, μ est une mémoire et ℓ est une nouvelle location.

– La règle pour l'instruction call est :

$$(S \cdot (g, pc, \mathbf{u} \cdot v_1 \cdots v_n), \mu) \rightarrow (S \cdot (g, pc + 1, \mathbf{u}) \cdot (f, 1, v_1 \cdots v_n), \mu)$$

où $g[pc] = (\mathsf{call}\ f\ n)$.

Exercice 9.2 *Formaliser la sémantique de toutes les instructions décrites ci-dessus.*

9.2 Compilation

– Soit w une liste de variables. $i(x, w)$ est la position la plus à droite de x dans w. Par exemple, $i(x, y \cdot x \cdot z \cdot x \cdot w) = 4$.
– On compile le corps principal du programme avec w liste vide et les corps des procédures avec w égal à la liste des paramètres formels.
– On compile les expressions comme suit :

$$\begin{array}{ll}
\mathcal{C}(n, w) & = (\mathsf{build}\ n\ 0) \\
\mathcal{C}(x, w) & = (\mathsf{load}\ i(x, w)) \\
\mathcal{C}(op(e_1, \ldots, e_n), w) & = \mathcal{C}(e_1, w) \cdots \mathcal{C}(e_n, w) \cdot (\mathsf{op}\ n) \\
\mathcal{C}(Pair(e_1, e_2), w) & = \mathcal{C}(e_1, w) \cdot \mathcal{C}(e_2, w) \cdot (\mathsf{build}\ Pair\ 2) \\
\mathcal{C}(Fst(e), w) & = \mathcal{C}(e, w) \cdot (\mathsf{fst}) \\
\mathcal{C}(!e, w) & = \mathcal{C}(e, w) \cdot (\mathsf{read}) \\
\mathcal{C}(ref\ e, w) & = \mathcal{C}(e, w) \cdot (\mathsf{new})
\end{array}$$

– On compile une liste de déclarations de variables comme suit (on omet le *let* dans les déclarations) :

$$\mathcal{C}(y_1 = e_1; \ldots; y_n = e_n, w) = \mathcal{C}(e_1, w) \cdots \mathcal{C}(e_n, w \cdot y_1 \cdots y_{n-1})$$

– La compilation des commandes nécessite un paramètre additionnel κ qui représente l'adresse de l'instruction où il faut sauter pour poursuivre le calcul. Nous faisons l'hypothèse que chaque segment de code de procédure contient une instruction return dont l'adresse est dénotée symboliquement par κ_{return}. De même nous supposons que le segment du code principal contient une instruction stop dont l'adresse est dénotée symboliquement par κ_{stop}.

– Considérez la compilation de la séquentialisation de deux commandes comme, par exemple, $(\mathit{if}\ x\ \mathit{then}\ S_{11}\ \mathit{else}\ S_{12}); S_2$. Ici on doit exécuter une des branches S_{1j} et ensuite procéder à l'exécution de S_2. A cette fin, on a besoin de connaître l'adresse de la première instruction du code qui correspond à la commande S_2. D'autre part, la valeur exacte de cette adresse va dépendre du nombre d'instructions produites par la compilation des branches S_{11} et S_{12}. Pour ne pas alourdir la notation nous allons indiquer de façon symbolique la première adresse de la suite d'instructions qui correspond à la commande S_2. Si on écrit :

$$\nu\kappa'\ \mathcal{C}(S_1, w, \kappa')\ \kappa' : \mathcal{C}(S_2, w, \kappa)$$

il est entendu que κ' est une nouvelle adresse à partir de laquelle on mémorise le code associé à la commande S_2. Par ailleurs, en passant κ' comme paramètre à la compilation de la commande S_1, on s'assure qu'une fois le calcul de S_1 terminé, le calcul va continuer avec l'instruction d'adresse κ'.

– Avec les conventions qu'on vient de présenter, on compile le corps principal $y_1 = e_1; \ldots; y_n = e_n; S$ par

$$\mathcal{C}(y_1 = e_1; \ldots; y_n = e_n, w) \cdot \mathcal{C}(S, w \cdot y_1 \cdots y_n, \kappa_{stop})$$

où w correspond aux variables prédéfinies.

– On compile le corps d'une procédure

$$\mathit{procedure}\ f(x_1 : \tau_1, \ldots, x_m : \tau_m) = y_1 = e_1; \cdots y_n = e_n; S$$

par

$$\mathcal{C}(y_1 = e_1; \ldots; y_n = e_n, x_1 \cdots x_m) \cdot \mathcal{C}(S, x_1 \cdots x_m \cdot y_1 \cdots y_n, \kappa_{return})$$

– Enfin on doit définir la compilation d'une commande.

$$\begin{array}{ll}
\mathcal{C}(x := e, w, \kappa) & = \mathcal{C}(e, w) \cdot (\mathsf{load}\ i(x, w)) \cdot (\mathsf{write}) \cdot (\mathsf{goto}\ \kappa) \\
\mathcal{C}(S_1; S_2, w, \kappa) & = \nu\kappa'\ \mathcal{C}(S_1, w, \kappa')\ \kappa' : \mathcal{C}(S_2, w, \kappa) \\
\mathcal{C}(f(e_1, \ldots, e_n), w, \kappa) & = \mathcal{C}(e_1, w) \cdots \mathcal{C}(e_n, w) \cdot (\mathsf{call}\ f\ n) \cdot (\mathsf{goto}\ \kappa) \\
\mathcal{C}(\mathit{while}\ e\ \mathit{do}\ S, w, \kappa) & = \nu\kappa'\ (\kappa' : \mathcal{C}(e, w) \cdot (\mathsf{branch}\ \kappa) \cdot \mathcal{C}(S, w, \kappa')) \\
\mathcal{C}(\mathit{if}\ e\ \mathit{then}\ S_1\ \mathit{else}\ S_2, w, \kappa) & = \mathcal{C}(e, w)\ \nu\kappa'\ (\mathsf{branch}\ \kappa') \cdot \mathcal{C}(S_1, w, \kappa)\ \kappa' : \mathcal{C}(S_2, w, \kappa)
\end{array}$$

Exemple 9.3 *On calcule la compilation du programme :*

$$\begin{array}{l}
\mathit{procedure}\ f\ (x : \mathit{int}, y : \mathit{ref}\ \mathit{int}) = \\
\mathit{if}\ x \neq 0\ \mathit{then}\ y :=\!!y + x; f(x - 1, y) \\
\mathit{let}\ z = \mathit{ref}\ 0; \\
f(3, z)
\end{array}$$

Ici on utilise une nouvelle commande if e then S. Elle peut être vue comme une abréviation pour la commande if e then S else skip où skip est une commande qui n'a pas d'effet. On préfère utiliser une compilation optimisée :

$$\mathcal{C}(\mathit{if}\ e\ \mathit{then}\ S, w, \kappa)\ \ = \mathcal{C}(e, w) \cdot (\mathsf{branch}\ \kappa) \cdot \mathcal{C}(S, w, \kappa)$$

Dans la compilation, on remplace aussi tout code de la forme :

$$\begin{array}{ll}
n : & \mathsf{goto}\ n + 1 \\
n + 1 : & \mathsf{instruction}
\end{array}$$

par le code : n : instruction. *Avec ces conventions on obtient :*

main	1 :	build 0 0	*f*	1 :	load 1
	2 :	new		2 :	build 0 0
	3 :	build 3 0		3 :	neq 2
	4 :	load 1		4 :	branch 16
	5 :	call f 2		5 :	load 2
	6 :	stop		6 :	read
				7 :	load 1
				8 :	add 2
				9 :	load 2
				10 :	write
				11 :	load 1
				12 :	build -1 0
				13 :	add 2
				14 :	load 2
				15 :	call f 2
				16 :	return

On remarquera que l'instruction 16 qui suit l'appel de fonction est un return. *On est donc en présence d'une récursion terminale et on pourrait remplacer les instructions 15 et 16 par* 15 : tcall f 2, *tout en remplaçant l'adresse 16 par l'adresse 15 dans l'instruction de saut conditionné 4.*

Exercice 9.4 *On considère un fragment de la compilation des commandes du langage impératif :*

$$
\begin{array}{ll}
\mathcal{C}(x := e, w, \kappa) & = \mathcal{C}(e, w) \cdot (\mathsf{load}\ i(x, w)) \cdot (\mathsf{write}) \cdot (\mathsf{goto}\ \kappa) \\
\mathcal{C}(S_1; S_2, w, \kappa) & = \nu\kappa'\ \mathcal{C}(S_1, w, \kappa')\ \kappa' : \mathcal{C}(S_2, w, \kappa) \\
\mathcal{C}(\textit{while } e \textit{ do } S, w, \kappa) & = \nu\kappa'\ (\kappa' : \mathcal{C}(e, w) \cdot (\mathsf{branch}\ \kappa) \cdot \mathcal{C}(S, w, \kappa')) \\
\mathcal{C}(\textit{if } e \textit{ then } S_1 \textit{ else } S_2, w, \kappa) & = \mathcal{C}(e, w)\ \nu\kappa'\ (\mathsf{branch}\ \kappa') \cdot \mathcal{C}(S_1, w, \kappa)\ \kappa' : \mathcal{C}(S_2, w, \kappa)
\end{array}
$$

où certaines adresses sont traitées de façon symbolique. Dans cet exercice, on souhaite définir une nouvelle fonction de compilation où les adresses sont toujours calculées explicitement.

1. *Définissez une fonction size(S) qui calcule le nombre d'instructions présentes dans la compilation de S (on suppose que la fonction size(e) est déjà définie sur les expressions).*
2. *Utilisez la fonction size pour définir une fonction de compilation (sans adresses symboliques !) $\mathcal{C}(S, w, i, \kappa)$ qui compile la commande S par rapport à une liste de variables w, en sachant que l'adresse de la première instruction du code compilé est i et que l'adresse de la première instruction à exécuter après S est κ.*
   Par exemple, si size$(e) = 1$, alors la fonction $\mathcal{C}(x := e; y := e, w, 12, 33)$ pourrait être une liste d'instructions de la forme suivante :

   $$\mathcal{C}(e, w) \cdot (\mathsf{load}\ i(x, w)) \cdot (\mathsf{write}) \cdot (\mathsf{goto}\ 16) \cdot \mathcal{C}(e, w) \cdot (\mathsf{load}\ i(y, w)) \cdot (\mathsf{write}) \cdot (\mathsf{goto}\ 33)$$

   qui est mémorisée entre les adresses 12 et 19.

9.3 Erreurs et typage du code octet

Comme dans l'évaluation du code source, on peut rencontrer un certain nombre de situations anormales ou erreurs, dans l'exécution du code octet. Par exemple :

– On affecte au compteur ordinal une valeur qui dépasse la taille du segment de code.
– On cherche une valeur sur la pile qui n'est pas présente.

– Les types des arguments sont incompatibles avec l'opération qu'on veut effectuer (projeter un entier, lire un booléen,...)
– On appelle une procédure qui n'existe pas ou on appelle une procédure avec le mauvais nombre d'arguments.

Comme pour le code source, on peut envisager une analyse de typage au niveau du code octet. Même si une analyse de typage est effectuée au niveau du code source, une telle analyse peut être utile pour au moins deux raisons :
– Le compilateur peut introduire des erreurs de type.
– Le code octet peut avoir été manipulé.

Comment typer un code octet ? L'idée est de calculer pour chaque instruction les types des valeurs qui peuvent être sur la pile de valeurs quand l'instruction est exécutée.

Exemple 9.5 *Pour chaque instruction du code octet généré dans l'exercice 9.3, on peut essayer de prévoir le nombre et le type des valeurs présentes sur la pile quand l'instruction est exécutée. Par exemple, pour le segment f on peut calculer :*

f	1 :	load 1	$int, ref(int)$
	2 :	build *0* 0	$int, ref(int), int$
	3 :	neq 2	$int, ref(int), int, int$
	4 :	branch 16	$int, ref(int), bool$
	5 :	load 2	$int, ref(int),$
	6 :	read	$int, ref(int), ref(int)$
	7 :	load 1	$int, ref(int), int$
	8 :	add 2	$int, ref(int), int, int$
	9 :	load 2	$int, ref(int), int$
	10 :	write	$int, ref(int), int, ref(int)$
	11 :	load 1	$int, ref(int)$
	12 :	build -1 0	$int, ref(int), int$
	13 :	add 2	$int, ref(int), int, int$
	14 :	load 2	$int, ref(int), int$
	15 :	call f 2	$int, ref(int), int, ref(int)$
	16 :	return	$int, ref(int)$

Remarque 9.6 *Le problème de la définition d'une machine virtuelle et de la génération de code pour le langage à objet discuté en section 7 peuvent constituer la base pour un mini-projet de compilation. Il s'agit d'adapter les concepts déjà présentés pour le langage impératif.*

Exercice 9.7 *On considère la commande S suivante :*

$$\begin{array}{l} \mathit{while}\ !x > 0\ \mathit{do} \\ \quad x := !x - !y; \\ \quad y := !y + !y \end{array}$$

Proposez une compilation de la commande S dans le code octet. La compilation de S est relative à une liste de variables $x \cdot y$ et à une continuation κ.

Exercice 9.8 *Décrivez la fonction de compilation de la commande repeat présentée dans l'exercice 6.16. Vous pouvez utiliser les instructions* goto *et* branch *dont la description suit :*
– goto j *: on affecte j au compteur ordinal.*
– branch j *: si la valeur au sommet de la pile est true on incrémente le compteur ordinal sinon on affecte le compteur ordinal à j. Dans les deux cas on supprime la valeur au sommet de la pile.*

Exercice 9.9 *On se place dans le cadre (d'un fragment) du langage impératif auquel on ajoute les commandes skip, fail et catch(S, S'). La syntaxe des commandes est spécifiée par la grammaire ci-dessous :*

$$S \ ::= \ id := n \mid S; S \mid skip \mid fail \mid catch(S, S)$$

L'évaluation des commandes est décrite par un jugement de la forme $(S, \eta, \mu) \Downarrow (X, \mu')$ où μ, μ' sont des mémoires et $X \in \{fail, skip\}$ indique si le calcul termine normalement (skip) ou si une exception (non-capturée) a été levée (fail). Les règles d'évaluation sont les suivantes :

$$\frac{X \in \{fail, skip\}}{(X, \eta, \mu) \Downarrow (X, \mu)} \qquad \frac{}{(x := n, \eta, \mu) \Downarrow (skip, \mu'[n/\eta(x)])}$$

$$\frac{(S_1, \eta, \mu) \Downarrow (skip, \mu') \quad (S_2, \eta, \mu') \Downarrow (X, \mu'')}{(S_1; S_2, \eta, \mu) \Downarrow (X, \mu'')} \qquad \frac{(S_1, \eta, \mu) \Downarrow (fail, \mu')}{(S_1; S_2, \eta, \mu) \Downarrow (fail, \mu')}$$

$$\frac{(S_1, \eta, \mu) \Downarrow (skip, \mu')}{(catch(S_1, S_2), \eta, \mu) \Downarrow (skip, \mu')} \qquad \frac{(S_1, \eta, \mu) \Downarrow (fail, \mu') \quad (S_2, \eta, \mu') \Downarrow (X, \mu'')}{(catch(S_1, S_2), \eta, \mu) \Downarrow (X, \mu'')}$$

Appliquez les règles d'évaluation aux commandes ci-dessous à partir d'un environnement η_0 qui associe aux variables a, b, c, d les locations distinctes $\ell_1, \ell_2, \ell_3, \ell_4$ et une mémoire μ_0 qui associe aux locations ℓ_i, $i = 1, 2, 3, 4$, la valeur 5.

$$S_1 = catch(a := 1; fail; b := 2 \ , \ c := 3)$$
$$S_2 = catch(a := 1; fail \ , \ catch(b := 2; fail; c := 3 \ , \ d := 4))$$

On ajoute une instruction fail *qui arrête le calcul dans un état d'échec (alors que l'instruction* stop *arrête le calcul avec succès).*

Définissez une fonction de compilation $\mathcal{C}(S, w, \kappa, \kappa')$ où S est la commande à compiler, w est une liste de variables, κ est l'adresse à laquelle continuer le calcul si l'évaluation de S termine normalement et κ' est l'adresse à laquelle continuer le calcul si l'évaluation de S lève une exception.

Calculez la compilation des commandes S_1 et S_2 décrites ci-dessus avec paramètres, $w = abcd$, $\kappa = 100$, $\kappa' = 200$ et en supposant que la première instruction est mémorisée à l'adresse 1.

Exercice 9.10 *On se place dans le cadre du langage à objets étudié dans la section 7. On considère les expressions suivantes :*

$$e ::= id \mid \mathsf{new}\ C(e, \ldots, e) \mid e.f$$

où id est la catégorie syntaxique des identificateurs et f celle des attributs. L'objectif est de définir une fonction de compilation $\mathcal{C}(e, w)$, où w est une liste d'identificateurs, pour une machine virtuelle qui s'inspire de celle étudiée pour le langage impératif. Dans la suite on rappelle et, au passage, on adapte certaines instructions de la machine virtuelle. Un bloc d'activation a la forme $(\ldots, pc, u_1, \ldots, u_m)$ où pc est le compteur ordinal et u_i dénote soit une valeur $C(\ell_1, \ldots, \ell_n)$ d'un objet soit une location ℓ.

- build *C n : on remplace n locations $\ell_1 \cdots \ell_n$ au sommet de la pile par la valeur $C(\ell_1, \ldots, \ell_n)$ et on incrémente le compteur ordinal. Ici C est le nom d'une classe.*
- load *n : on copie l'n-ième élément de la pile au sommet de la pile et on incrémente le compteur ordinal.*

- prj j *: si l'élément au sommet de la pile est une valeur* $\mathsf{D}(\ell_1, \ldots, \ell_n)$ *avec* $1 \leq j \leq n$ *alors on remplace cette valeur par* ℓ_j *et on incrémente le compteur ordinal.*
- new *on génère une nouvelle location* ℓ*, on écrit la valeur au sommet de la pile dans la location, on remplace la valeur par* ℓ *au sommet de la pile et on incrémente le compteur ordinal.*
- read *si la valeur au sommet de la pile est une location* ℓ*, on remplace* ℓ *par son contenu et on incrémente le compteur ordinal.*

Donnez les règles pour la compilation des expressions.
Générez le code associé à l'expression $e = (\mathsf{new}\ C(\mathsf{new}\ D(), x)).f$*, par rapport à la liste d'identificateurs* $w = x \cdot y \cdot x$ *où l'on sait que* f *correspond au premier attribut de la classe* C*.*

10 Gestion de la mémoire

Le code exécutable généré par un compilateur est un processus qui tourne au dessus d'un système d'exploitation. Le processus dispose d'un certain segment de mémoire virtuelle qui doit être géré de façon économique.

Les machines virtuelles des langages de programmation courants (C, Java, ML,...) distinguent trois zones de mémoire : une zone statique qui contient le code, les variables globales, les tampons d'entrée-sortie,... une pile (ou *stack*) qui contient la pile des blocs d'activation des procédures et un tas (ou *heap*) qui contient des données dont la taille ou la durée de vie ne sont pas prévisibles.

La gestion de la pile est simple. Il suffit de garder un pointeur au sommet de la pile. Pour allouer un bloc de b cellules on incrémente le compteur de b en vérifiant qu'il n'y a pas de débordement, pour enlever un bloc de b cellules on décrémente le pointeur de b.

La gestion du tas est plus compliquée. Le problème est de déterminer le moment auquel on peut récupérer une certaine partie de la mémoire. Plusieurs options ont été considérées :

- On ne récupère jamais la mémoire. Ceci est correct mais peut produire une saturation de la mémoire.
- Le programmeur indique explicitement quand une cellule peut être disposée. C'est l'option prise par C mais elle a des inconvénients majeurs : on peut oublier de récupérer une cellule et pire on peut récupérer une cellule qui est encore accessible avec des conséquences catastrophiques sur le comportement du programme.
- On analyse statiquement le programme pour déterminer les régions du tas qui peuvent être récupérées. Cette approche a été experimentée dans un langage de la famille ML (ML-Kit) mais elle n'est pas encore très répandue.
- La machine virtuelle appelle un programme dit ramasse miettes (ou *garbage collector*) pour récupérer les cellules inaccessibles. C'est l'option choisie par les langages modernes comme ML et Java et c'est l'option sur laquelle nous allons nous concentrer.

Le problème d'écrire un bon ramasse miette est encore le sujet de recherches. Nous allons juste considérer trois méthodes de base.

La mémoire est modélisée par un graphe dirigé avec racines. Les noeuds du graphe sont les cellules de la mémoire, les arêtes dirigées représentent les pointeurs et les racines sont les cellules dans la zone statique et dans la pile. Une cellule dans le tas est *récupérable* si elle n'est pas accessible à partir des racines.

Au début du calcul toutes les cellules libres du tas sont connectées dans une liste. Quand une nouvelle cellule est nécessaire, on extrait un élément de la liste. S'il n'y a plus de cellules disponibles dans la liste, on peut interrompre l'exécution du programme et appeler le ramasse miettes pour vérifier si une partie de la mémoire du tas peut être récupérée et insérée à nouveau dans la liste des cellules libres.

Dans la suite on suppose que toutes les cellules ont la même taille. En général il faut considérer l'allocation de cellules de taille variable (par exemple pour l'allocation de tableaux).

10.1 Marquage et balayage (mark and sweep)

On suppose que toutes les cellules comprennent un bit de marquage qui est initialement à 0. La méthode de marquage et balayage fonctionne en deux phases.

Marquage On visite le graphe en commençant par les racines et on met à 1 les bits de marquage de toutes les cellules accessibles.

Balayage On va parcourir toutes les cellules du tas et pour chaque cellule on effectue les opérations suivantes :
- Si le bit de marquage est à 0 alors on insère la cellule dans la liste des cellules libres.
- Si le bit de marquage est à 1 on le remet à 0.

La phase de marquage est normalement effectuée par une visite en profondeur d'abord du graphe :

```
Init :  sp := nil;

procedure DF(v)
if v  points to heap and v.mark = 0 then
   begin
   push(v, sp);
   while sp ≠ nil do
      begin
      v := pop(sp);
      v.mark := 1;
      ∀w(w pointer in cell v and w.mark = 0) do
          push(w, sp);
      end
   end
```

On doit appeler la procédure $DF(v)$ sur chaque adresse v du tas qui est contenue dans une cellule dans la zone statique ou dans la pile.

Exercice 10.1 *Comment peut-on modifier les structures de données de cet algorithme pour qu'il visite le graphe* en largeur *?*

Rappel : considérez un arbre binaire avec racine 1 dont les fils sont 2 et 3, et tel que les fils de 2 sont 4 et 5 et les fils de 3 sont 6 et 7. Dans une visite en profondeur *(de gauche a droite) on visite les noeuds dans l'ordre 1,2,4,5,3,6,7 alors que dans une visite* en largeur *on visite les noeuds dans l'ordre 1,2,3,4,5,6,7.*

La phase de ramassage est implémentée simplement. On suppose que *fl* pointe à la liste des cellules libres du tas.

```
Init :  p := 'lower address of heap';

   while p < 'upper address of heap' do
      begin
      if p.mark = 1 then p.mark := 0
      else insert(p, fl);
       p := p + 'cell size';
      end
```

Le coût de la méthode de marquage et ramassage est déterminé facilement. Soit R le nombre de cellules dans le tas qui sont accessibles à partir de la zone statique et de la pile. Soit H le nombre total de cellules disponibles dans le tas. Alors le coût est donné par :

$$c_1 R + c_2 H$$

pour des facteurs constants c_1 et c_2, où $c_1 R$ est le coût du marquage qui est proportionnel au nombre de cellules accessibles et $c_2 H$ est le coût du ramassage du tas.

Il est intéressant de considérer le coût par cellule récupérée. Il est exprimé par

$$(c_1 R + c_2 H)/(H - R)$$

car $(H - R)$ est exactement le nombre de cellules récupérées. On remarque que si $H \approx R$ alors le coût est élevé et que si $H >> R$ alors le coût s'approche de c_2.

Ceci suggère que il n'est pas très intéressant d'exécuter la méthode de ramasse miettes quand une grande partie du tas est accessible. Dans ce cas, la machine virtuelle passe son temps à essayer de récupérer un nombre réduit de cellules. Quand cette situation se vérifie, la machine virtuelle peut essayer d'obtenir de la mémoire virtuelle additionnelle du système d'exploitation.

10.2 Comptage des références (reference counting)

Un problème avec la méthode de marquage et ramassage est que son exécution provoque l'arrêt de l'exécution du programme pour un temps proportionnel à la taille du tas. Cet arrêt peut être inacceptable pour des programmes qui doivent respecter des contraintes de temps réel. La méthode du comptage des références règle partiellement ce problème. Voici les ingrédients de la méthode :

- Chaque cellule du tas comprend un champ compteur qui compte le nombre de pointeurs à la cellule.
- Initialement le compteur est à 0.
- Pour chaque instruction, le compilateur génère un certain nombre d'instructions qui maintiennent le compteur à jour.

Par exemple, considérons la situation suivante :

cell address	env field in cell	counter field in cell
x	y	n_1
y	*nil*	n_2
p	*nil*	n_3

On suppose que le programme comprend l'instruction $x.env := p$. Le compilation doit générer la séquence suivante d'instructions où, comme dans la section précédente, on suppose que *fl* pointe à la liste de cellule libres dans le tas.

$x.env.counter := x.env.counter - 1;$
if $x.env.counter = 0$ *then*
  begin
  $insert(x.env, fl);$
  recursively update counters of cells pointed by $x.env$
  end
$x.env := p;$
$p.count := p.count + 1$

Maintenant la gestion de la pile est interlacée avec l'exécution du programme. On dit que la méthode du comptage de références est une méthode de ramasse miettes *incrémentale*. On peut remarquer un certain nombre de limites de la méthodes :

1. Comme illustré dans l'exemple, la méthode est coûteuse.
2. La mise à jour récursive des compteurs des cellules pointées par $x.env$ peut prendre un certain temps. Si nécessaire, on peut suspendre cette mise à jour et la reprendre plus tard.

3. On ne récupère pas toujours la mémoire disponible.

Pour illustrer le dernier point, on suppose être dans la configuration :

cell address	env field in cell	counter field in cell
r	p	1
p	x	1
x	y	2
y	x	1

avant d'exécuter le code associé à l'affectation $r := nil$. Maintenant considérons le code généré. Comme r pointe à p, on décrémente $p.counter$. Comme $p.counter$ va à 0 on récupère p. De plus, comme p pointe à x on décrémente $x.counter$ qui va à 1. Maintenant ni x ni y sont accessibles mais ces cellules ne peuvent pas être récupérées car leurs compteurs ne sont pas à 0. Le point est que le comptage de références ne voit pas l'inaccessibilité de structures *avec cycles* et donc il peut ne pas récupérer des cellules qui ne sont plus accessibles. En pratique, un ramasse miettes qui utilise le comptage des références fait aussi appel périodiquement à une autre méthode de ramasse miettes.

10.3 Récupération par copie (copying collection)

La méthode de marquage et ramassage a deux problèmes additionnels :

– Pour marquer le graphe on a besoin d'une pile dont la taille est bornée par le nombre de cellules du tas. Donc on peut avoir besoin de beaucoup de mémoire juste au moment où la mémoire est épuisée.[5]

– La mémoire récupérée peut être de plus en plus fragmentée ce qui est un problème si on a besoin d'allouer des données de taille variable sur des blocs de cellules contiguës.

Ces deux problèmes sont réglés par la méthode de *récupération par copie* que nous allons présenter.

Le tas est maintenant divisé en deux moitiés composées de cellules contiguës. On appelle la première moitie 'from_space' et la deuxième 'to_space'.

Initialement, on alloue dans la zone 'from_space'. Quand cette zone est saturée, le ramasse miette traverse la partie accessible de 'from_space' et construit une copie isomorphe dans un segment initial de la zone 'to_space'. L'algorithme qui traverse les noeuds accessibles et qui génère la copie isomorphe est la partie intéressante de la méthode.

La première fois qu'on arrive à une cellule accessible de la zone 'from_space' on copie son contenu dans la première cellule disponible dans la zone 'to_space'. La cellule dans la zone 'from_space' est alors marquée et un pointeur à sa copie dans la zone 'to_space' est inséré. Le marquage est important pour éviter que la cellule soit recopiée plusieurs fois. On remarque que le problème de la fragmentation dans la zone 'to_space' a disparu.

Une fois que la phase de copie est complétée, on ne procède *pas* à une phase de ramassage. En effet, il suffit d'invertir simplement le rôle de 'from_space' et de 'to_space' et de continuer l'exécution du programme. Ceci veut dire que si R est le nombre de cellules accessibles dans la zone 'from_space' alors le coût de la méthode est cR pour une constante c et le coût par cellule récupérée est $cR/((H/2) - R)$. Si $H >> R$ alors le coût approche 0, mais en pratique R est plutôt proportionnel à H.

[5]Nous verrons qu'il y a une méthode dite d'inversion des pointeurs qui, plutôt qu'utiliser une pile, utilise les cellules accessibles pour effectuer une visite en profondeur.

On décrit maintenant l'algorithme qui copie la partie accessible de 'from_space' dans 'to_space'. On suppose que chaque cellule contient un champ $f1$. Il peut s'agir d'un champ spécial ou du premier champ de la cellule s'il y en a un. On suppose que *next* et *scan* sont deux pointeurs qui pointent initialement à l'adresse de base de 'to_space'. D'abord on doit définir une procédure *Fwd* qui va créer une copie s'il n'y en a pas déjà une.

function $Fwd(p) = case$
p points to from_space and $p.f1$ points to to_space :
  $p.f1$

p points to from_space and $p.f1$ does not point to to_space :
  $copy(p, next)$;
  $p.f1 := next$;
  $increment(next)$;
  $p.f1$

else : p

Soit r la racine du graphe 'from_space'. On exécute :

$Fwd(r)$; (this increments *next*)

while $scan < next$ *do*
  begin
  $\forall$ pointer field f in the cell pointed by *scan* *do*
    $scan.f := Fwd(scan.f)$;
  $increment(scan)$;
  end

Un point intéressant de l'algorithme est qu'il n'utilise pas de mémoire additionnelle pour visiter le graphe dans 'from_space' (ce qui n'était pas le cas pour la méthode de marquage et ramassage). La raison est que les éléments à visiter sont mémorisés dans la zone 'to_space' entre les pointeurs *scan* et *next*.

Exemple 10.2 *En exécutant la méthode de ramassage par copie sur l'exemple :*

Address	*Field* $f1$	*Field* $f2$
7	9	11
9	7	9
11	9	7

et en supposant que la racine r *est* 7 *et que l'adresse de base de 'to_space' est* 12 *on produit la copie suivant dans la zone 'to_space' :*

Address	*Field* $f1$	*Field* $f2$
12	13	14
13	12	13
14	13	12

Exercice 10.3 (1) *Déterminez sous quelles conditions le coût en temps d'exécution d'une méthode de* mark and sweep *est inférieur à celui d'une méthode de* copying collection.

(2) *Un collègue, suggère d'intercaler l'exécution du programme principal et de la procédure de* mark and sweep *qui de cette façon opèrent comme deux processus concurrents et interruptibles qui agissent sur une structure partagée (la mémoire). Expliquez à votre collègue les problèmes qui pourraient se présenter suite à une interruption du processus de* mark and sweep *(on vous demande de déterminer des problèmes, pas de trouver des solutions...)*

Exercice 10.4 *On dispose d'un tableau qui contient des blocs de taille variable. Si p est l'adresse du premier mot d'un bloc alors on dénote avec $p.statut$ son statut qui peut être libre ou occupé et avec $p.long$ sa longueur. Décrivez un algorithme linéaire dans la taille du tableau qui permet de compacter la mémoire, c'est-à-dire de faire en sorte que les blocs occupés sont contigus et précèdent un bloc libre. Voici un exemple de tableau avant et après compactage où X dénote des informations non significatives mémorisées dans les blocs libres et $a, b, c, \ldots$ dénotent des informations significatives mémorisées dans les blocs occupés.*

	Avant		*Après*
1 :	$(libre, 2)$	1 :	$(occupé, 2)$
2 :	X	2 :	a
3 :	$(occupé, 2)$	3 :	$(occupé, 3)$
4 :	a	4 :	b
5 :	$(libre, 1)$	5 :	c
6 :	$(occupé, 3)$	6 :	$(libre, 5)$
7 :	b	7 :	X
8 :	c	8 :	X
9 :	$(libre, 1)$	9 :	X
10 :	$(libre, 1)$	10 :	X

10.4 Inversion de pointeurs

Nous présentons une méthode pour visiter en profondeur un graphe qui n'utilise pas une pile mais qui demande de réserver un petit nombre de bits pour chaque cellule du tas.

Pour simplifier, on suppose que chaque cellule pointée par x contient deux pointeurs au tas qu'on désigne par $x.f0$ et $x.f1$. En plus, chaque cellule contient un champ d'un bit *mark* et un champ de 2 bits *done* (en général, le nombre de bits dans ce champ est logarithmique dans le nombre de pointeurs au tas dans la cellule).

```
local current, pred, next, i;
current := root; current.done := 0; current.mark := 1; pred := nil;
while true do
i := current.done;
if i < 2
then
    next := current.fi
    if next.mark = 0
    then
        current.fi := pred; pred := current; (1)
        current := next; current.mark := 1; current.done := 0;
    else
        current.done := i + 1;
else
    next := current;
    current := pred;
    if current = nil then STOP;
    i := current.done;
    pred := current.fi;
    current.fi := next; (2)
    current.done := i + 1;
```

Dans (1), $current.fi$ est sauvé dans *next* et il pointe ensuite à la cellule d'où *current* a été accédé. Dans (2), la valeur originale de $current.fi$ est restaurée. Le tableau suivant décrit la

suite de valeurs contenus dans les pointeurs et les noeuds quand l'algorithme est exécuté sur le graphe $G = \{(1,2),(1,4),(2,3),(2,4),(3,1),(3,2),(4,1),(4,3)\}$.

current	$1, 2, 3, 2, 4, 2, 1, nil$ $(\rightarrow$ STOP$)$
next	$2, 3, 2, 1, 3, 4, 3, 1, 4, 2, 4, 1$
pred	$nil, 1, 2, 1, 2, 1, nil$

	mark	done	f_0	f_1
1	$0, 1$	$\_, 0, 1, 2$	$2, nil, 2$	4
2	$0, 1$	$\_, 0, 1, 2$	$3, 1, 3$	$4, 1, 4$
3	$0, 1$	$\_, 0, 1, 2$	2	1
4	$0, 1$	$\_, 0, 1, 2$	3	1